

PISA

A Programming System for Interactive Production of Application Software

HABILITATIONSSCHRIFT

zur Erlangung der venia legendi an der
Rechts- und staatswissenschaftlichen Fakultät
der Universität Zürich

von

Dr. Rudolf Marty

von Malans (Kt. Graubünden)

1980



Digitized by the Internet Archive
in 2026

PISA

A Programming System for Interactive Production of Application Software

HABILITATIONSSCHRIFT

zur Erlangung der *venia legendi* an der
Rechts- und staatswissenschaftlichen Fakultät
der Universität Zürich



von

Dr. Rudolf Marty

von Malans (Kt. Graubünden)

A219

A Programming System for Interactive
Simulation of Application Systems

UNIVERSITY OF LEIDEN

LIBRARY OF THE
UNIVERSITY OF LEIDEN
LEIDEN, THE NETHERLANDS



ON THE
UNIVERSITY OF LEIDEN

To Rebekka

Preface

This report introduces the programming system PISA intended for the interactive production of application software. The heart of the programming system is a new programming language, also called PISA. An interactive environment for this language permits the creation, test, maintenance, and usage of PISA programs in a real-time dialogue fashion. Both the programming language and its interactive environment are described without any reference to a specific implementation. Together, they form a well defined programming system whose components interact harmoniously.

The programming system PISA is dedicated to application software production. This implies that production of system software and online-control programs is not a goal of PISA. Furthermore, it means that PISA must meet several requirements as they arise from commercial application software production, the most stringent ones being economy, availability, and compatibility: In the long term the overall cost of software production and usage with such a programming system must be less than with conventional means, the programming system must be available or implementable on a wide range of computer systems currently used, and existing data must be accessible in its actual physical representation.

The definition of PISA is given in a heavily annotated form in this report: Examples for the use of single components as well as for the entire system are presented, most of the language constructs and system facilities are commented on briefly, and the implications of the programming system's design on implementability and portability issues are discussed.

It was one of the main objectives for PISA's design to base the programming system on well-known concepts that have proven to be useful and implementable. New ideas whose consequences on the programming system's structure and implementation are not completely understood were excluded from PISA.

Table of contents

Section A	explains the reasons for defining a comprehensive programming system and gives the design principles for an interactive programming language as well as for an interactive programming system.
Section B	defines the programming language PISA, the predefined routines, and the interfaces between a PISA program and its environment.
Section C	describes the syntax and the semantics of a PISA session providing the interactive environment for creating, testing, maintaining, and using PISA programs.
Section D	shows how the programming system PISA can be employed in several typical areas of applications. Particular attention is paid to the use of the PISA interfaces.
Section E	gives some guidelines for an implementation of PISA and suggests efficient run-time code structures for several language constructs and system components.
Section F	focuses on the portability aspects of PISA programs: Following a discussion of design aspects that make PISA a highly portable language, a portability checker for PISA programs and a validation system for PISA implementations is proposed.

A preface and a table of contents appear on the first pages of every section. Where a keyword index to a section is useful, it is included as one of the section's appendices. All chapter and appendix numbers are prefixed with a letter designating the section membership.

Acknowledgements

This report has been written during a most beneficial and enjoyable stay with the Computer Systems Research Group at the University of Toronto. I am greatly indebted to the Computer Systems Research Group for providing me an excellent environment for my work and to the "Erziehungsrat des Kantons Zuerich" for the grant that made this stay affordable for my family.

I am thankful to Prof. Niklaus Wirth for the programming language Pascal and his permission to use portions of the Pascal report. The authorization to copy parts of the Pascal report as given by Springer-Verlag is also appreciated. Pascal was a very advantageous starting-point for my work on the language PISA.

Both the aspects of the programming system PISA and the wording of this report have profited from a number of people who gave me valuable comments at various stages of PISA's design. In this regard I am very obliged to James Cordy, David Crowe, David Gries, Eric Hehner, Richard Holt, Thomas Hull, Hugh Redelmeier, Leonard Vanek, and David Wortman.

I would particularly like to thank Inge Weber who did an expert job on typing and correcting the manuscript with the text editing system ATS (IBM). Without her help the preparation of the manuscript would have taken considerably more time.

Toronto, Canada
August 1978

Rudolf Marty

SECTION A

The Design of an Interactive Programming System
for Production of Application Programs

As an introduction to the definition of the programming system PISA, this section describes the fundamental criteria for designing an interactive programming system devoted to the production of application programs. After a discussion of the reasons for expanding the definition of a programming language to the definition of an entire programming system, the basic design principles for the programming language and its surrounding programming system are stated.

Table of contents

WHY A PROGRAMMING SYSTEM?	A-1.
The domain of a programming language definition	A-1.1.
Interactive programming and testing	A-1.2.
Language front end processors	A-1.3.
FUNDAMENTAL LANGUAGE CHARACTERISTICS	A-2.
Basic grammatical structure	A-2.1.
Data abstraction	A-2.2.
Algorithmic abstraction	A-2.3.
Block structure and lifetime of data objects	A-2.4.
Human and economic factors	A-2.5.
PRINCIPLES OF AN INTERACTIVE PROGRAMMING SYSTEM	A-3.
Operational abstraction	A-3.1.
Program text manipulation	A-3.2.
User program execution	A-3.3.
Testing aids	A-3.4.
Human and economic factors	A-3.5.

A-1. WHY A PROGRAMMING SYSTEM?

=====

The term "programming system" has been coined for the interactive software production tool to be defined in this report. Before we can specify the design guidelines for an interactive programming system, we must answer the questions: "Why a programming system? Is a programming language not all we need?" or, alternatively, "What is the difference between a programming language and a programming system?". The following paragraphs discuss the reasons for designing a comprehensive software production tool as a programming system and not simply as a programming language.

A-1.1. The domain of a programming language definition

The definition of a high level programming language describes an abstract model of computation. A set of syntax rules is given for the language, and precise semantics is associated with all phrases of the language. The language definition may leave certain rules to be specified by the implementation. Maximum range and precision of numbers, interfaces to the operating system, and the character set used are but a few of such implementation dependent language rules. The abstract language definition augmented with what we call "implementation parameters" forms a precise and complete description of a hypothetical machine. This hypothetical machine has a well defined initial state, accepts and executes a program written in its language, and, finally, comes to a normal or, if a language violation has been detected, abnormal terminal state.

The augmented language definition is a complete and sufficient statement for program creation and program verification. Together with the input data, it defines the output of a given program and ultimately guides our reasoning about the program. Yet, the language definition is far from being a complete and sufficient description of an interactive software production tool because it merely specifies the behaviour of the system during the time the hypothetical machine given by our language has control (this includes a compilation of the language). In order to make the language usable, we need an environment around it: we need a system to physically create and maintain our programs, we need diagnostics from the hypothetical machine that are clearly related to the actual program, we need a method that allows the inspection of objects defined by the program after it

has abnormally terminated, etc. It is this environment which makes a programming system out of a programming language. We shall discuss the various requirements for a programming system in the following chapters.

Clearly, the domain of a programming language definition is limited to the description of a "naked" hypothetical machine. The language definition specifies neither how the hypothetical machine is embedded in an enclosing system, nor how it can be externally controlled and monitored, nor in what form it emits diagnostics to its environment. Traditionally, it has been left to the implementor to make all these decisions which lie outside the domain of the language definition.

A-1.2. Interactive programming and testing

If the hypothetical machine given by a programming language definition is used in batch mode, the decisions left outside the language definition's domain are much less important than if it is embedded in an interactive system. In batch mode, the user has only two trivial interfaces to the programming system - the source program he enters and the listing he receives. There is no interaction between the user and the system during execution of the program. The most critical decisions to make by a batch implementation designer relate to the realm of language violation checking and reporting (we do not consider resource optimization of any kind here).

In interactive programming, on the other hand, the design of the components not described by the language definition have a tremendous influence on the productivity of the programming tool and, as a result, on the user satisfaction with it. To illustrate this, let us first enumerate some requirements to an interactive programming tool and then analyse their implications on its design:

- The language violation diagnostics (both syntax errors and semantic errors) are reported with an indication to the construct of the source program that caused the violation.
- The execution of a program is interruptable and dynamically traceable.
- After any kind of error or interruption during a program's execution, the program is stopped at a well defined

location. This location is reported to the user with respect to the source program.

- After a program has been stopped at the location reported, a given subset of the programming language can be entered for immediate execution. This means that the language text is entered and immediately executed. All objects of the stopped program that are in a defined scope are thereby accessible.
- The execution of a program can be resumed from the location it has been stopped at, i.e. the program regains control.

The requirements mentioned above are a subset of the specifications to an interactive software production tool as we shall discuss in A-2. and A-3. Clearly, all these requirements lie outside the domain of a programming language's definition. Nevertheless, the syntax and semantics of the dialogue introduced to meet them can only be defined in the context of the programming language. The language, in turn, must have a syntactic and semantic structure which facilitates the implementation of these requirements as an intuitive and consistent extension of the language.

We conclude that a tool for interactive software production has to be designed so that the definition of the programming language and the definition of the system implementing the interactive environment form a well defined programming instrument whose components interact harmoniously. We call such an instrument a "programming system" to denote its global definition as opposed to the restricted domain of the definition of a "programming language".

A-1.3. Language front end processors

Generally, a front end processor is a system of software and hardware managing the data flow between another processor (the back end processor) and some input/output facility. A well-known application of front end processors is interconnecting terminal lines and a main processor channel or bus, where the front end processor monitors the lines, translates code, and performs line protocol adaptation tasks. As the title of this chapter suggests, we use the term "front end processor" in connection with programming languages: We mean a processor translating a language into another language that is not executed yet but again

translated by a compiler. The terms "preprocessor" and "precompiler" are commonly associated with such processors, however, as we shall see, the term "front end processor" is much more accurate for their use in interactive programming systems.

Front end processors to various high level programming language compilers as COBOL, PL/I, and FORTRAN have become very popular in the past. They range in complexity from simple macro processors over moderately sophisticated language extension processors (e.g. decision table processors) to quite complex processors for translating nonprocedural languages or nonprocedural language extensions (e.g. simulation languages, program generators for sequential file processing applications, etc.).

Traditional language front end processors write the program they generate to an intermediate file which is then used as input to the compiler in a second step. After the generated program has been compiled to executable code, this code is executed in a third step (let us neglect linking and loading for the sake of simplicity). Hence, the user may obtain diagnostic messages from any of the three steps "front end processing", "compilation", and "execution". The compilation and execution diagnostics are related to the program generated by the front end processor and not to the original program entered by the user. As a consequence, the user must be able to map the program generated by the front end processor back into the original program in order to locate and correct compiletime and execution errors. This might be quite trivial if the original language is close to the generated one, for example if only simple macros have been expanded locally. On the other extreme, the mapping may be so complex that only a sound understanding of the front end processor's internal logic allows us to find a compile time or execution error in the original program. This becomes particularly troublesome if the original program has a nonprocedural structure.

What would be preferable is a front end processor that not only translates an input language into another language passed to the compiler, but also handles the information flow in the opposite direction: It maps the diagnostic information from the compilation and execution steps back to the original program, so the user is no longer concerned with the translation logic. We require that the programming language we propose be embedded in a programming system enhancing the implementation of such front end processors.

The conventional implementation of language front end processors as the first step in a sequential stream "front end processing - compilation - execution" is not applicable in interactive

programming and testing because of a reason analogous to the one given for the diagnostic messages: As we have seen in the preceding chapter (A-1.2.), the entire system dialogue during testing is closely related to the source program. This source program, of course, is the program generated by the front end processor not the one entered by the user of the interactive system. If the user in fact did all his tests based on the generated program, this would have two main consequences: Firstly, he could alter the generated program and change variables as well as the control flow by immediate statements during its test; secondly, the user would have to be able to map the generated program to the original program in order to understand the generated program and to make the necessary corrections to the original program.

While this might be feasible and desirable in simple front end processor applications, it is unthinkable for an interactive front end processor translating a complex nonprocedural language. In such cases, the distinction between front end processing and compilation should be completely transparent to the user. Hence, the front end processor is in charge not only of generating a program but also of processing the testing dialogue and "transrelate" it from the original program to the generated program and vice versa.

It should now be definitely clear why the term "front end processor" is preferable to "preprocessor" in an interactive environment: The front end processor handles the entire data flow (i.e. dialogue) between the user and the programming system, it does not simply generate a program.

A-2. FUNDAMENTAL LANGUAGE CHARACTERISTICS

=====

In this chapter, we shall declare the fundamental characteristics of a high-level programming language which may serve as a basis for an interactive programming system. The selection and formulation of the characteristics is intended to be consistent with two objectives: that the principles lead to a clean model of computation, and that the language be suitable for commercial application programming. The enumeration of the characteristics is neither exhaustive nor are the single characteristics specified in detail: The purpose of this chapter is to state principles not details.

A-2.1. Basic grammatical structure

The terminal symbols of the language are representable with the information codes used by the majority of computer systems. The symbols in the intersection between the ASCII code and the EBCDIC code fulfil this requirement.

The language is defined without any respect to line numbers associated with the program text. Where such numbers are used, they serve exclusively for identification purposes and are not denotable in the language itself.

The grammar of the language is context free and has a structure such that its sentences are parsable with at most a one-symbol lookahead and with only a single pass over the source text.

A-2.2. Data abstraction

The data types introduced by the language are not related to any specific implementation. Every data type defines the values objects of its type may assume and the set of operations which may be performed on them. The type of every data object is manifest from the program text. This implies that a data object has one and only one type during execution.

Let us use the term "simple types" for types that are not composed of other types (terms used synonymously are "scalar types", "elementary types", and "unstructured types"). The language defines simple types for representing numbers of a given range and precision, strings of a given number of characters, and the logic values "true" and "false". Other simple types may be introduced by the language.

The language provides two basic methods of structuring data objects into what we could call a "structured type". The first method allows to define an ordered collection of named data objects of arbitrary types; the data objects in the resulting structure are denoted by their names. A second method can be used to arrange data objects of a uniform type in an n-dimensional space; the data objects in this space are denoted by their coordinates. Other structuring methods may be defined by the language.

The language is type safe, i.e. any physical data space may only be denoted by a data object of the same type as the value the data space actually holds.

Data objects of any type may be declared to represent either a constant value which is manifest from the program text, or variable values as assigned to the data object during execution of the program.

The language's data abstraction concept enables the implementation to define a mapping from the abstract data types to their representation such that existing data bases established by programs of another language can also be used and updated by programs written in the proposed language.

A-2.3. Algorithmic abstraction

The language defines a set of simple statements which serve to assign values to data objects, to direct the flow of control, and to communicate with other parts of the program or its environment. A special simple statement may be introduced to denote no action. Structured statements specify sequential, conditional, or repetitive execution of their component statements. Any statement may be a component statement. Both the simple and the structured statements are defined without any reference to a specific implementation. Collectively, they form an abstract algorithmic model based on the language's data abstraction model.

Language text describing an algorithmic action may be encapsulated in a named frame, commonly called a "procedure". This algorithmic action is then denotable by the procedure's name both outside and inside the procedure itself. A procedure may accept parameters and may return a value. Optionally, a procedure can preserve the state of its data objects after control has been returned to the point of its denotation. The text of a procedure may be contained in the program or it may be external to it.

The language defines simple and transparent interfaces for communication between a program and its environment. These interfaces are denoted by a syntax which is consistent with the rest of the language. The definition of specific interfaces may be implementation dependent.

The flow of control in every program written in the language is serial and synchronous. No parallel execution and no asynchronous exception handling can be programmed.

A-2.4. Block structure and lifetime of data objects

The language is block structured, i.e. the text of a program is composed of a hierarchical structure of textually nested blocks such that the outermost block is the program itself. Each block may contain declarations of objects and executable statements. The objects declared in a block are denotable in this block and in all blocks nested within it, but not in surrounding blocks. The language may restrict this rule. The declarations and statements establishing blocks are defined by the language; at least the program and each procedure establishes a block in the sense described above.

If data objects declared in a block are called static data objects, then the lifetime of static data objects is the same as the lifetime of the block they are declared in. If more than one instance of the block exists, each instance has its own instance of static data objects associated with it.

Data objects can also be created and destroyed explicitly by the program if they are not static objects. If such data objects are called dynamic, then the lifetime of dynamic data objects is the time between their creation and their deletion, i.e. their lifetime is not related in any way to the textual program structure.

A-2.5. Human and economic factors

The language is easy to use. Its syntax and semantics can be acquired in a reasonable time. After an initial training period, the programmer completely understands the language. This understanding includes the ability to judge the resource requirement of the program and parts thereof.

Programs written in the language are self documenting. Minimizing the textual length of programs is not a goal of the language.

Programs written in the language are highly portable from one environment to another.

An efficient implementation of the language can be produced with an economically acceptable investment in manpower and computer resources.

A-3. PRINCIPLES OF AN INTERACTIVE PROGRAMMING SYSTEM

=====

In a similar way as chapter A-2. introduced the principles for the design of a programming language, this chapter describes the basic design guidelines for an interactive programming system surrounding this language. While we noted that the fundamentals given for the language are of a rather general nature not related to the language's interactive environment, we shall see that the structure of an interactive programming system has a major influence on the language's implementation. We shall also realize that an interactive programming system can only be defined with respect to a specific programming language which, in turn, must have a structure that makes it suitable as a basis for an interactive programming system.

This chapter does not address the global structure of a multi-user programming system and its interrelation with the operating system. These topics are too dependent on the specific operating system supporting the programming system's implementation. The principles stated concern the interaction between a user and the programming system.

A-3.1. Operational abstraction

The programming system assigns an abstract machine to every user connected to it. This abstract machine is defined without any reference to a particular implementation. It provides the software and hardware resources for interactively creating, testing, maintaining, and using application software. If physical resources are shared among several such abstract machines, the sharing is transparent to the users and has no consequences to the abstract machines' definition. In accordance with common terminology, we call such an abstract machine a "session".

The user is connected to the session by a bidirectional communication path. The communication path is restricted to serial transmission of characters. Apart from this path, no other communication links like control lines, interrupt lines, or similar exist between the user and the session. The message transfer over the communication path is called "dialogue".

The session accomplishes two basic tasks: First, it performs the functions defined by the programming system, and second, it executes user programs. In the former case, the syntax and the semantics of the dialogue between the user and the session are given by the definition of the programming system, in the latter case, they are given by the executing program and the definition of its programming language.

The user of a session can be any process, external or internal to the programming system. An external process is commonly a human user connected to the session via a terminal, it, however, could be an external hardware processor as well. An internal process acting as user is a program executing within the programming system and communicating with the controlled session via a software interface emulating the two-way communication path.

A-3.2. Program text manipulation

The text of a program is composed of a number of lines. Each line has a unique line number associated with it which serves to identify the line; the line number is not part of the program text itself. The ordered sequence of the program text lines is given by ascending line numbers independent of the order in which the lines have been added to the text.

The programming system defines the dialogue of an editor which allows to add new lines to an existing program text, to delete lines from it, to change the text of specific lines, to renumber the lines, and to inspect the program text. A special command of the editor serves to request a formatted printout of the entire program text.

Program text can be made subject to a syntax check. The syntax diagnostics are defined by the programming system; every syntax diagnostic message designates the textual location of the language construct that caused the error.

The physical representation of the program text is transparent to the user. The text of a specific program is identified by a user assigned name. If program text is to be preserved beyond the duration of the actual session, it can be stored under this name on secondary storage assigned to the user.

A-3.3. User program execution

After a program execution has been initiated by the user, the two-way communication path between the user and the session is automatically linked to the executing program. If this program does not maintain a dialogue with the user, it may disconnect the communication path and execute without a connection to any user. Otherwise, the communication path remains linked to the program during its entire execution.

The programming system introduces two modes of execution: A "test mode" enforces all rules of the programming language and reports any violation as soon as it occurs. The second mode is trimmed for execution speed and could be called "production mode". This mode merely catches errors that represent a danger to the safe and secure operation of other users and of the programming system as a whole.

The programming system defines the execution errors detected in each of the two modes, the resulting diagnostic messages, and the action taken after the error has been reported. Every execution diagnostic message designates the textual location of the language construct that caused the violation.

An executing program that has a connection to a user may be interrupted by its user. The programming system defines the effect such user interrupts have on the executing program and the action taken after the execution has been halted. The textual location where a program has been halted as a result of a user interrupt is reported to the user. Any program may enable and disable the user interrupts during arbitrary periods of its execution.

A-3.4. Testing aids

Testing aids are only defined for programs executing in what has been named "test mode" above. The "production mode" does not provide any testing facilities.

The flow of control of an executing program may be monitored by a tracing concept based on the language definition. The trace can be done in a single-step fashion where there is a pause in execution after each trace step, or continuous tracing can be requested.

Program errors and enabled user interrupts lead to a program halt at a well defined location within the program. This location is always set between two executable statements of the program and is reported to the user after the program has been halted.

The programming system defines the subset of the programming language constructs that may be entered for immediate execution after a program has been halted. The syntax and the semantics of the program text entered are defined by the programming language definition such that the text is considered to be virtually inserted at the location of the program halt.

A special command serves to resume the execution of a halted program from the location it has been halted at.

A-3.5. Human and economic factors

The interactive programming system is easy to use. The syntax and the semantics of the session dialogue can be learned thoroughly during the initial training period with its programming language. Thereafter, the user is able to take full advantage of all facilities provided for interactive programming and testing.

An efficient implementation of the programming system can be produced based on available hardware and system software with dialogue capabilities. The investment in the implementation is economically justified by the savings in software production and maintenance during the implementation's lifetime.

SECTION B

The Programming Language PISA

This section describes the programming language PISA. The language description includes examples and notes to promote the understanding of the constructs introduced. The structure of this section is designed to minimize the number of forward references designating as yet undefined language parts. Where such references are given, they usually can be ignored by a reader with a thorough knowledge of block structured languages as Pascal, Algol, Simula, and even PL/I.

PISA is based on Pascal. Some parts of the Pascal Report [Wirth,75] have been copied into the language description without any changes or with only minor modifications. Wherever Pascal is mentioned within this section, the reference is to [Wirth,75] and is not explicitly given.

The data type concept of PISA has been taken from Pascal. Several ambiguities of Pascal as described by [Welsh et al.,77] and [Haberman,73] are due to the type definitions given in the Pascal report. These ambiguities have been resolved in PISA by refined type definitions.

The major differences between PISA and Pascal are summarized below to provide a quick overview for a reader familiar with Pascal.

- The PISA basic vocabulary uses only characters that are in both the EBCDIC and ASCII character sets.
- PISA does not define any special statement separators or statement terminators. The semicolon is treated as a single character comment and may be inserted into a PISA program text wherever a blank is allowed.
- A variable cannot be defined to be of type integer but only of some subrange thereof.

- A real type definition must explicitly declare the range of values variables and constants of this type may assume and must indicate the desired precision.
- String types of fixed or varying length are simple types.
- Variant records allow any suitable field within the record's fixed part to be a tag field. Variant records have the same case syntax as the case statement.
- File types are not supported in PISA.
- Simple and structured constants may be declared.
- An identifier can be bound to any existing variable and denotes this variable in the scope of its binding declaration. Binding of a variable to a record variable supersedes Pascal's "with" statement.
- Value constructors allow to create a value of any structured type.
- In PISA, the boolean operators "not", "and", and "or" have a lower precedence than the arithmetic and the relational operators.
- Procedures, functions, compound statements, and repetitive statements each constitute a block; conditional statements constitute one or more blocks. Each block denotes a scope and consists of a declaration part followed by a statement part.
- Variable, binding, constant, type, procedure, and function declarations may appear in arbitrary order within the declaration part of a block. As a rule, however, identifiers must be declared before they are used.
- Labels are identifiers.
- A multilevel exit statement allows the termination of named blocks.
- PISA defines only one repetitive statement - the loop statement. Together with the multilevel exit statement, it replaces Pascal's for, repeat, and while statements.
- Procedures and functions can be coroutines. A suspend statement suspends the execution of a coroutine.

- Procedures and functions can be compiled separately and denoted as external procedures or external functions, resp.
- A conditional statement may be a decision table.
- Procedures and functions cannot be parameters in PISA.
- A PISA program communicates with its environment solely via two interfaces, called "user interface" and "system interface".

Table of contents

THE LEXICAL STRUCTURE OF PISA	B-1.
Notation and vocabulary	B-1.1.
Identifiers, numbers, and literal strings	B-1.2.
Condition entries and action entries	B-1.3.
Separators and comments	B-1.4.
Termination of syntactic units	B-1.5.
DATA TYPES	B-2.
Type declarations	B-2.1.
Simple types	B-2.2.
Discrete types	B-2.2.1.
Enumerated types	B-2.2.1.1.
Predefined enumerated types	B-2.2.1.2.
Subrange types	B-2.2.1.3.
Real types	B-2.2.2.
String types	B-2.2.3.
Structured types	B-2.3.
Array types	B-2.3.1.
Record types	B-2.3.2.
Set types	B-2.3.3.
Pointer types	B-2.4.
Type compatibility	B-2.5.
Type adaptation	B-2.5.1.
Type conversion	B-2.5.2.
CONSTANTS AND VARIABLES	B-3.
Constant declarations	B-3.1.
Denotation of constants	B-3.2.
Variable declarations	B-3.3.
Denotation of variables	B-3.4.
Binding	B-3.5.
EXPRESSIONS	B-4.
Value constructors	B-4.1.
Operators	B-4.2.
Sign operators	B-4.2.1.
Exponentiation operator	B-4.2.2.
Multiplying operators	B-4.2.3.
Adding operators	B-4.2.4.
Relational operators	B-4.2.5.
Boolean operators	B-4.2.6.
Function designators	B-4.3.

BLOCKS AND SCOPES OF IDENTIFIERS	B-5.
Blocks	B-5.1.
Scope of identifiers	B-5.2.
STATEMENTS	B-6.
Label declarations	B-6.1.
Simple statements	B-6.2.
Assignment statements	B-6.2.1.
Procedure statements	B-6.2.2.
Goto statements	B-6.2.3.
Exit statements	B-6.2.4.
Suspend statements	B-6.2.5.
Structured statements	B-6.3.
Compound statements	B-6.3.1.
Conditional statements	B-6.3.2.
If statements	B-6.3.2.1.
Case statements	B-6.3.2.2.
Decision table statements	B-6.3.2.3.
Repetitive statements	B-6.3.3.
PROCEDURE DECLARATIONS	B-7.
Procedure definitions	B-7.1.
External procedures	B-7.2.
Forward procedure declarations	B-7.3.
FUNCTION DECLARATIONS	B-8.
Function definitions	B-8.1.
External functions	B-8.2.
Forward function declarations	B-8.3.
PROGRAMS AND SUBROUTINES	B-9.
Programs	B-9.1.
Subroutines	B-9.2.
PREDEFINED PROCEDURES AND FUNCTIONS	B-10.
Ordinal functions	B-10.1.
String handling functions	B-10.2.
Numeric functions	B-10.3.
Conversion and editing functions	B-10.4.
Dynamic storage procedures	B-10.5.
Program control procedures	B-10.6.
Miscellaneous procedures and functions	B-10.6.
INTERFACES OF A PISA PROGRAM	B-11.
The user interface	B-11.1.
The system interface	B-11.2.

Appendix B-I: Collected syntax of PISA

Appendix B-II: System service requests

Appendix B-III: Programming example

Appendix B-IV: Keyword index

Appendix B-V: References

B-1. THE LEXICAL STRUCTURE OF PISA

B-1.1. Notation and vocabulary

The syntax of PISA is described in a slightly modified Backus-Naur form. Syntactic constructs are denoted by English words or phrases written in lower case letters and not enclosed in any special brackets. The name chosen for a construct is intended to suggest its meaning. Terminal symbols are enclosed in quotes ". An underscored double quote "_" is a double quote belonging to the terminal symbol. The curly brackets { and } denote possible repetition of the enclosed construct or alternatives of constructs zero or more times. Possible omission of a construct or alternatives of constructs is shown by enclosing it within square brackets [and]. Embedding several alternatives within parenthesis (and) indicates that exactly one of the alternatives has to be selected.

The structure of the grammar defining the syntax of PISA is not intended for parsing the language. Its purpose is to describe the language in a user-oriented form as a supplement to the written text. An SLR(1) parser grammar for PISA is given in E-1.1.

The basic vocabulary of PISA consists of basic symbols classified into letters, digits, and special symbols.

```
letter      ::= "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K" |
              "L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V" |
              "W"|"X"|"Y"|"Z" |
              "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k" |
              "l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v" |
              "w"|"x"|"y"|"z" |
              "_"|"_"|"_"
```

```
digit       ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
```

```
spec_symbol ::= "+" | "-" | "*" | "/" | "***" | "=" |
              "<" | "<" | ">" | "<=" | ">=" | ">" |
              "(" | ")" | "." | "(" | "(" | "(" | "=" |
              "." | "." | ":" | ":" | ":" | ":" | ":" |
              ">" | "<:" | ">:" | "(:" | "(:" | "(:" |
word_symbol
```

```
word_symbol ::= "action"|"and"|"area"|"array"|"begin"|"bind"|
               "case"|"cond"|"const"|"coroutine"|"datab"|"div"|
               "downto"|"else"|"end"|"exit"|"external"|"fix"|
               "float"|"for"|"forward"|"function"|"goto"|
               "if"|"in"|"loop"|"main"|"mod"|"nil"|"not"|
               "of"|"or"|"otherwise"|"packed"|"procedure"|
               "record"|"rep"|"set"|"string"|"suspend"|
               "then"|"to"|"type"|"var"|"vary"|"when"|"while"
```

Capital letters within a word symbol are treated as small letters. Thus, capital letters can be used in word symbols without changing their meaning.

The characters selected to form the basic vocabulary of PISA are all contained in the standard ASCII character set as well as in EBCDIC. This enhances portability of PISA source programs. The following Pascal symbols have been replaced:

Pascal	PISA
-----	-----
{	(*
}	*)
[	(and (:
]	) and :)
up arrow	->

B-1.2. Identifiers, numbers, and literal strings

Identifiers are used to denote constants, variables, types, labels, blocks, procedures, and functions.

```
identifier ::= letter {letter | digit | "_"}
```

The break character "_" serves to visually decompose an identifier into its component parts. Two identifiers are equal if and only if they correspond character by character. A small letter corresponds to its capital equivalent when used in an identifier. An identifier may be between 1 and 16 characters long. The word symbols defined in B-1.1. may not be used as identifiers; the identifiers "integer", "boolean", "true", "false", and "char" may not be declared.

```

Examples:      balance      Left_Successor
               ACCT_#       SLOT_1
               City         %_InterestRate

```

Whenever a name of a syntactic construct in PISA's grammar ends with "_id", it is an identifier:

```
..._id      ::= identifier
```

where "..." denotes any sequence of characters.

The common decimal notation is used for numbers. The syntax of unsigned numbers is summarized in the rules:

```

integer      ::= digit {digit}
number       ::= integer "." integer |
               integer ["." integer]
               ("E"|"e") ["+"|"-" ] integer

```

```

Examples:      5      7.391      0.015      17.3E-7      1.526e17

```

Sequences of characters enclosed in single quote marks are called literal strings. A literal string may contain all characters available on the implementation. A single quote as part of the literal string is represented by two consecutive single quotes. A literal string may not cross the boundaries of a program text line and can include between 0 and 255 characters.

```
lit_string    ::= "'" {character} "'"
```

```

Examples:      'Q'      'ok?'      'PETER O'TOOLE'      '' (=nullstring)
               'This is an example of a longer literal string'

```

B-1.3. Condition entries and action entries

Condition and action entries are used in decision table statements (B-6.3.2.3.). They consist of a sequence of rule tokens enclosed in double quotes " (represented as " in the productions below). An entry may contain between 1 and 32 rule tokens.

```

cond_entry    ::= "-" {"Y"|"y"|"N"|"n"|"-" } ""
action_entry  ::= "" {"X"|"x"|"-" } {"X"|"x"|"-" } ""

```

```

Examples:      "-yN-N-Yn--"
               "XXX--x--"

```

E-1.4. Separators and comments

Blanks, semicolons, tabulation characters (e.g. ASCII 9), end-of-lines, and comments are considered to be separators. An arbitrary number of separators may be inserted between any two successive PISA symbols except within the ones defined in E-1.1., E-1.2., and E-1.3. At least one separator must occur between all pairs of consecutive identifiers, numbers, or word symbols. Except for end-of-lines, separators within a literal string are part of the literal string. An end-of-line illegally terminates the literal string.

Comments are enclosed in comment brackets (* and *) and may contain any sequence of characters available on the implementation except the sequence *) which is used to terminate a comment. A comment may cross the boundaries of an arbitrary number of program text lines.

E-1.5. Termination of syntactic units

PISA program text consists of a sequence of declarations and statements. Each statement and declaration is composed of one or more syntactic units. A syntactic unit is not terminated by any special termination symbol (as e.g. ; or end-of-line). For the convenience of PISA programmers familiar with the use of semicolons in other languages, the semicolon is a separator in PISA. Thus, it can be inserted into PISA program text at places where other languages require it without changing the meaning of the program.

B-2. DATA TYPES

=====

Many high level languages provide "problem" data types which are bound to hardware structures. Declaring an integer variable in Pascal, for example, is not only based on the programmer's intention to use this variable for storing whole numbers, but also on his knowledge of the integer's machine representation. This because the machine representation defines the range of values a variable of type integer may assume. Similarly, the programmer relies on the internal representation of real types in Pascal, for their accuracy and range is implicitly given by the number of bits the hardware uses for mantissa and exponent.

PISA does not include any data types directly or indirectly involving a knowledge of their internal representation. Each type definition determines the values which variables of that type may assume and the set of basic operations which may be performed on them. This allows the compiler either to find a suitable representation for the type or to reject the type definition, saying that the underlying implementation is not capable of supporting it. Whenever internal representation of data types is treated in this report, it is only of concern to the implementor of a PISA system or to a highly skilled PISA programmer who needs this information for using data bases established by non-PISA programs.

B-2.1. Type declarations

A type declaration associates an identifier with the type. This identifier is called a type identifier and is considered to denote the same type as its definition. A type identifier is known within the current scope and must be declared before it is used. Scope rules will be treated in B-5.2. and are not a prerequisite for the understanding of type declarations.

```
type_dcl      ::= "type" type_id "=" type_def
type_def      ::= simple_type | struc_type | pointer_type |
                  type_id
```

A root type will be defined together with every type t. It is the type from which t is derived. If t is not derived from any other type, t is its own root type.

If a type `t` is declared by

```
type t = u
```

where `u` is a type identifier, `t` is the same type as `u` (and, of course, has also the same root type).

B-2.2. Simple types

A simple type is an unstructured type.

```
simple_type ::= discrete_type | real_type | string_type
```

B-2.2.1. Discrete types

Variables and constants of a discrete type may only assume the discrete values introduced in their type's definition. A discrete type is either an enumerated type or a subrange type.

```
discrete_type ::= enum_type | subrange_type
```

B-2.2.1.1 Enumerated types

An enumerated type defines an ordered set of discrete values by enumeration of the identifiers which denote these values. They are declared as constants of this type within the current block, where they may not be used for any other purpose. An enumerated type is its own root type.

```
enum_type ::= "(" constant_id {"," constant_id} ")"
```

```
Examples:  type colour = (red,blue,green,yellow,white)
           type day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun)
           type employment = (junior_progr,progr,senior_progr)
           type sex = (male,female)
           type delivery = (by_mail,by_truck,by_train,taken_out)
```


B-2.2.1.2. Predefined enumerated types

The following enumerated types are predefined for every PISA program.

```
type integer = (...,-3,-2,-1,0,1,2,3,...)
```

The values of type integer are the positive and negative integers in the mathematical sense. It is not possible to declare a variable to be of type integer; it must be declared to be of some problem-determined subrange thereof. The predefined type declaration for integer introduces its constants, namely signed integers as -89532781102, - 15, 0, 37, +472863, defined by:

```
integer_const ::= ["+"|"-"] integer
```

An integer constant written without a sign is assumed to be positive. The type integer explicitly defines only one value zero. For hardware with "one's complement" or "sign and magnitude" arithmetic using both -0 and +0, the implementation provides the necessary instructions to change all occurrences of -0 to +0 before the program can access the value.

```
type boolean = (false,true)
```

The constants of the type boolean are the truth values denoted by the identifiers "false" and "true". The word "boolean" has become a technical term and, hence, is written in small letters (as are ohms, amperes, etc.).

```
type char = ( <enum> )
```

where <enum> stands for the enumeration of the characters determined by the character set of the implementation. The enumeration contains all characters of the implementation, printable and nonprintable, ordered in ascending sequence of their internal representation. A single character enclosed in quotes denotes a constant of this type. Examples: '1' '#' '""' 'A' (B-1.2.).

The identifiers "integer", "boolean", "char", "true", and "false" may not be declared. This makes sure that all language definitions involving one of these identifiers hold within any PISA program.

E-2.2.1.3. Subrange types

A type may be defined as a subrange of another already defined enumerated type, called its root type. The definition of a subrange simply indicates the least and the largest value in the subrange. The lower bound must not be greater than the upper bound.

```
subrange_type ::= entire_const ".." entire_const
```

The root type of the subrange type is the root type of the first constant in a subrange type definition and must be a discrete type. The second constant must be of the same root type as the first one.

```
Examples:  type workday = Mon..Fri      (*subrange of day*)
           type index = -10..+25        (*subrange of integer*)
           type letter = 'A'..'Z'       (*subrange of char*)
           type branch = 1..35          (*subrange of integer*)
```

For the programmer, subrange types provide a convenient means of expressing his knowledge about the values a variable of this type will assume. It is therefore strongly recommended that a subrange of integer is not defined with respect to the implementation, but always with respect to the underlying problem [Wortman,77].

Subranges of the root type "char" are implementation dependent. Care has to be taken that the specified subset covers the desired set of characters. Assuming EBCDIC code for example, the definition

```
type letter = 'A'..'Z'
```

not only includes the letters 'A' to 'Z', because the set of letters is not contiguous in EBCDIC.

E-2.2.2. Real types

A real type defines a subrange of the real numbers. Since, generally, real numbers cannot be represented exactly in digital computers, a real type definition must include a specification of the desired precision for this type.

```

real_type      ::= "fix" "(" range "," entire_const ")" |
                  "float" "(" range "," entire_const ","
                      entire_const ")"
range           ::= entire_const ".." entire_const

```

All real types explicitly define only one value zero. For hardware with "one's complement" or "sign and magnitude" arithmetic using both -0 and $+0$, the implementation provides the necessary instructions to change all occurrences of -0 into $+0$ before the program can access the value.

A fixed point type is defined by "fix(a..b,f)". a and b denote constants of root type real or integer and determine the lower and the upper bound of the covered subrange ($a < b$). f must be a constant of root type integer and specifies the precision by indicating the desired number of fractional decimal digits ($f \geq 0$) to be maintained for values of this type. a and b may not specify more than f fractional digits.

A floating point type is introduced by "float(a..b,m,z)", where a, b and z denote constants of root type real or integer. a specifies the lower and b the upper bound of the subrange for this type ($a < b$). Floating point values are implemented by tuples (mantissa,exponent). m is a constant of root type integer and determines the precision of a floating point type by specifying the desired number of decimal digits in the mantissa. z must be greater than zero and fixes the range of floating point values to be treated as zero such that a value x of this type is virtually zero if and only if $-z < x < +z$. In other words, z is the smallest absolute value of this type that has to be distinguished from zero. The highest exponent of a floating point type is given by the range declaration a..b, the lowest exponent by the value of z. a, b, and z may not exceed the precision given by m.

```

Examples:  type weight = fix(0..100,3)
           type angle  = fix(-45..+45,0)
           type dollar = fix(-1E12..+1E12,2)
           type voltage = fix(-0.75..120,2)
           type deflection = float(-2..+2,7,1E-7)
           type distance = float(0..1E20,5,0.000001)
           type stdflt = float(-1E30..+1E30,12,1E-29)

```

The declared precisions are always considered as minimum figures. The implementation is free to maintain greater accuracy. The rules governing operations on real types will be defined in B-4.2. The root type of fix and float types is "real".

Constants of real type are optionally signed numbers as 15378.9, +3.14159, - 1.5832E+7, 0.00052, 153E-3 and are defined by:

```
real_const ::= ["+"|"-" ] number
```

The precision of a real constant is determined based upon the number of fractional digits, respectively the total number of digits in the mantissa, if it is written in mantissa-exponent format.

One might argue that a fixed point type as used for certain units like money, length, weight, etc., is actually a discrete type and not a real (or analog) one. In fact, the discrete nature of such figures is only artificial because certain conditions enforce the usage of given units. Nobody can, for example, pay exactly for 784.29 grams of apples priced at 95 cents per kilogram; the commonly applied procedure is to round the inherent analog amount of money to the nearest cent. Nevertheless, money should be considered as an analog or real type. Discrete types are principally used for ordinal and cardinal purposes.

E-2.2.3. String types

A sequence of characters used to represent a text is called a string. A string type defines the maximum length of text that variables of its type can hold. The length is counted in number of characters. Each character in a string is a value of the predefined enumerated type "char". The root type of string types is "string".

```
string_type ::= "string" "(" entire_const ")" [ "vary" ]
```

The constant must be of root type integer and specifies the maximum length of text for this type. The maximum length is implementation dependent; string types up to 255 characters, however, are supported by every implementation of PISA. When the option "vary" is included in a string type definition, variables of this type hold texts of varying length. The actual length is determined upon assignment of a text and can be between 0 and the maximum length given in the type definition. An actual length of 0 indicates that the variable holds no text at all. Variables of a string type without the "vary" option always hold a text of their type's maximum length, padded with trailing blanks if necessary.

Examples: type name = string(30) vary
 type alpha = string(maxlen)
 type message = string(120) vary

Constants of type string are called literal strings and have been defined in B-1.2. A literal string containing exactly one character is a constant of type "char". The type conversion rules treated in B-2.5., however, allow us to consider a one-character literal string as a constant of type string as well. This interpretation is not very exact but does not alter the meaning of a program.

Pascal's definition of strings as "packed array [1..n] of char" has been given up in favor of a new simple type "string" similar to PL/I's character strings for several reasons. Informally, the definition of strings as arrays of characters seems incorrect because, generally, every element of an array represents a value independent from all other elements. The information stored in an array is hardly ever processed in its entirety. It is treated on an element-by-element basis. Obviously, a string of characters, a text, is not read this way; it makes only sense as a whole, certainly not by isolated inspection of single characters. Yet, the issue has also its formal aspects. The requirements of string handling make necessary the definition of several operations and functions not applicable to other array types but "packed array [1..n] of char". Examples are assignment between arrays of different index types, padding and truncation, designation of subsections of arrays (substrings), concatenation of arrays, and so forth. Both, the formal and the informal arguments seem to be enough reason for defining a separate simple type for strings together with the operations on it.

E-2.3. Structured types

A structured type is characterized by the type(s) of its components and by its structuring method. Moreover, a structured type definition may contain an indication of the preferred data representation: If a definition is prefixed with the symbol "packed", this is a hint to the compiler that storage should be economized even at the price of some loss in efficiency of access, and even if this may expand the code necessary for implementing access to components of the structure. A structured type is always its own root type. Hence, even if the type definitions of two structured types are textually identical they have distinct root types.

```
struc_type    ::= ["packed"] (array_type | record_type | set_type)
```

Components of a packed structured type cannot be used as variable or area parameters (B-4.3., B-6.2.2.) and cannot be renamed by a binding declaration (B-3.5.). Apart from these restrictions, the symbol "packed" does not change the meaning of a PISA program.

B-2.3.1. Array types

An array type is a structure consisting of a fixed number of components which are all of the same type, called the component type. The elements of the array are designated by indices, values belonging to the index type. The array type definition specifies the component type as well as the index type(s).

```
array_type    ::= "array" "(" index_type {"," index_type} ")"
                  "of" compnt_type
index_type    ::= discrete_type | type_id
compnt_type   ::= type_def
```

Index types must be discrete types other than integer. If n index types are specified, the array type is called n -dimensional, and a component is designated by n indices.

Examples: type array (1..100) of fix(0..1E6,2)
 type array (day,branch,empl_*) of (ill,holi,on_duty)
 type array (-5..+5,colour) of samples

(These examples assume the previous declaration of the index type identifiers as discrete types).

B-2.3.2. Record types

A record type is a structure consisting of a fixed number of components, possibly of different types. The record type definition specifies for each component, called a field, its type and an identifier which denotes it. The scope of these so-called field identifiers is the record definition itself. They are also accessible within a field constant or field variable (B-3.) referring to a record of this type.

A record type may have several variants. They are enumerated in a case construct terminating the optionally empty fixed part of a

record type. The selector in the case construct is called the tag. It is either a field identifier declared in the record's fixed part and indicates which variant is assumed by the record variable at a given time, or it is a type identifier specifying the data type of the constants in the when lists. The tag must designate a discrete type other than integer. Each variant begins with a when construct, which specifies a set of constants of the tag's type, or with an otherwise clause.

The when lists must be disjoint. Furthermore, the union of the lists must be equal to the set of constants defined for the tag type unless there is an "otherwise" variant. An "otherwise" variant comprises all tag values in the set difference S between the set of the constants defined for the tag type and the union of the when lists, i.e., it lumps all tag values not mentioned explicitly. If there is an otherwise variant and S is empty, the program is illegal.

```
record_type ::= "record"
              [ fixed_part ]
              [ variant-part ]
              "end" "record"
fixed_part  ::= {field_dcl}
variant_part ::= "case" tag
              {when_vnt}
              [ otherwise_vnt ]
              "end" "case"
when_vnt    ::= "when" when_list ":" {field_dcl}
otherwise_vnt ::= "otherwise" ":" {field_dcl}
when_list   ::= when_item {"," when_item}
when_item   ::= entire_const ["." entire_const]
field_dcl   ::= field_id {"," field_id} ":" type_def
tag         ::= field_id | type_id
```

A when item $m..n$ denotes all elements i in the subrange $m..n$ of the tag's type such that $m \leq i \leq n$.

```
Examples:  type date = record
            day:      1..31
            month:    (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,
                      Sep,Oct,Nov,Dec)
            year:     1900..2100
            end record
```

```

type tree_el = record
  el_type:      (node,leaf)
  case el_type
    when node:
      left_succ,right_succ: el_addr
      comp_value: inv_#
    when leaf:
      inventory_#: inv_#
      product: string(20) vary
  end case
end record

type transaction = record
  rec_type:      (bill,cred_note,
                  pmt_cash,pmt_bank)
  branch:        1..50
  transaction_date: date
  customer:      1..100000
  case rec_type
    when bill:
      invoice_#:      10000..99999
      amount_invoice: dollar
    when cred_note:
      text:           string(50)
      amount_credit:  dollar
    otherwise:
      currency:       curr_code
      date_paid:      date
      payment_code:   1..20
      amount_paid:    fix(0..1E6,2)
  end case
end record

```

All field identifiers defined within a record must be distinct, even if they appear in different variants.

The textual range of a when variant is closed by the succeeding when variant, by the otherwise variant, or by the end of the variant part ("end case"), whichever occurs first. An otherwise variant is closed by the end of the variant part ("end case"). Although the construct "end case" is redundant because it is always followed by "end record", it has been left in the definition of variant parts to use the same syntax as for the case statement (B-6.3.2.2.).

The definition of variant parts allows the specification any field of the record's fixed part as a tag for the variant part. This conforms to record structures of existing data bases where the tag field is not always the field immediately preceding the

variant part. In opposition to Pascal, the PISA case clause never declares a field; it includes only a field reference or a type reference. Hence, the Pascal program segment

```
... case ms: maritalstatus of ...
```

is rewritten in PISA as

```
... ms: maritalstatus
    case ms ...
```

B-2.3.3. Set types

A set type defines the range of values which is the powerset of its so-called base type, i.e., the set of all subsets of values of the base type, including the empty set. The base type must be a discrete type other than integer.

```
set_type      ::= "set" "of" base_type
base_type     ::= discrete_type | type_id
```

```
Examples:    type days = (mon,tue,wed,thu,fri,sat,sun)
              type week = set of days
              type symptoms=(headache,stomach_ache,nausea,vertigo,
                             diarrhoea,dry_skin,unconsciousness,
                             fever,high_pulse)
              type disease = set of symptoms
```

Sets can be built up from values of their base type as described in B-4.1. The programmer should be aware that a PISA implementation limits the size of sets. However, a minimum of 256 elements in a set type is provided by every PISA implementation.

B-2.4. Pointer types

Variables declared in a block (B-3.3.) are referred to by their identifiers. They exist during the entire lifetime of this block. Therefore, these variables are called static. In contrast, variables may also be allocated and freed dynamically, i.e. without correlation to the structure of the blocks. Since such a dynamic variable does not occur in an explicit variable declaration, it cannot be designated by an identifier. Instead,

access is achieved via a pointer which holds the address of the variable. The address of a dynamic variable is obtained upon its allocation (B-10.5.) and is stored in a pointer type variable. A pointer type thus consists of an unbounded set of addresses pointing to elements of the same type, called the pointer's associated type.

A pointer's root type is "pointer to t", where t is the pointer's associated type. The pointer constant "nil" points to no dynamic variable at all and belongs to every pointer type.

```
pointer_type ::= ">" assoc_type
assoc_type  ::= type_id
```

```
Examples:  type truck = record
              truck_type:  string(50)
              capacity:    kilos
              truck_driver: ->driver
            end record
type driver = record
              driver_name:  string(30)
              social_ins_#: number
              driving:      ->truck
            end record
type driverptr = ->driver
```

The associated type of a pointer must not necessarily be known when the pointer type is declared. It can be declared anywhere in the declaration part of the current block. This exception from the general rule that type identifiers must be declared before they are used allows direct or indirect recursive references. The above example illustrates such an application of pointers.

B-2.5. Type compatibility

A value of type t is compatible with type q if

- t and q are the same type, or if
- a value of type t is adaptable to type q, or if
- a value of type t is convertible to type q.

Two enumerated, structured, or pointer types t and q are the same type if they have the same root type. Two subrange, real, or string types t and q are the same type if they define the same values, or, in other words, if their definition is identical after all constant denotations have been replaced by the proper value.

A value of type t is adaptable to type q if (1) t and q have the same root type, and (2) the adaptation rules given in B-2.5.1. are not violated.

There are two pairs of mutually convertible root types: integer / real and char / string. The rules for their conversion are given in B-2.5.2.

E-2.5.1. Type adaptation

A value n of a subrange type t is always also a legal value of t's root type. Hence, if q is t's root type, n is adaptable to q regardless of the actual value n. If q is a subrange type with the same root type as t (t being a subrange or the root type itself), a value n of type t is adaptable to type q if and only if n is defined for q.

A real value x of type t is adaptable to a real type q if and only if x lies within the range of values defined for q. If the actual value of x is not representable exactly by a value of type q, then x is rounded to the nearest value defined for type q.

If t and q are string types, let s be the value of type t, k be the length of s, and m be the maximum length defined for q. If k > m, the value of type q is the string containing the first m characters of s. Otherwise, if q is a string type defined with the "vary" option, s becomes the value of type q, whose actual length is set equal to k; if q is defined without the "vary" option, the value of type q is s padded to its right with m-k blanks.

B-2.5.2. Type conversion

A value i of root type integer is convertible to a type q of root type real if and only if i lies within the range of values defined for q . If the actual value of i cannot be represented exactly by a value of type q , then i is rounded to the nearest value defined for type q .

A value x of a real type t is converted to a type q of root type integer by rounding. If the rounded value lies outside the range specified for q , the program is illegal.

The result of a char to string conversion is a string of length 1 containing the character.

The resulting character c from a string to char conversion is the character represented by the string s . If s has a length unequal to 1, the program is illegal.

B-3. CONSTANTS AND VARIABLES

B-3.1. Constant declarations

A constant declaration introduces a constant identifier denoting the constant. The constant declaration must textually precede any use of the constant identifier. A constant is a simple constant if it denotes a value of a simple type, it is a structured constant if it defines a structured type together with the values of its components, or it is a pointer constant if it denotes the pointer value "nil".

```
constant_dcl ::= "const" constant_id "="
               (entire_const | type_def const_list)
const_list   ::= "(:" [const_item {"", " const_item"}] ":)"
const_item   ::= entire_const [".." entire_const] | const_list
```

A constant identifier denoting an entire constant has the same type as the entire constant.

If the value of a structured constant is declared by a constant list, the type of the structured constant is explicitly given by the type definition preceding the constant list. This type definition must denote a structured type, of course. The constants in the constant list become the values of the components of the structured constant. They must be entire constants defined for the type of the component they are associated with. A constant item of the form *m..n* denotes a set of elements and is only allowed in a constant list for a set constant.

For an array constant, the order of the constants in the constant list is the linearized order of the array components. The linearized order of a one-dimensional array is obtained by selecting all its components in ascending order of their index values. The linearized order of an *n*-dimensional array is produced by rewriting it as *n* nested one-dimensional arrays and recursively applying the procedure for one-dimensional arrays (row-major).

For a record constant, the order of the constants in the constant list is the order in which the components appear in the record type definition. The number of constants in the constant list must be equal to the number of record components. A record constant may not contain a variant part.

A set constant contains all elements enumerated in the constant list. The constants in the constant list must be of the set's base type. An empty constant list denotes an empty set. A constant item of the form $m..n$ denotes the set of all elements i in the subrange $m..n$ of the base type such that $m \leq i \leq n$. If $m > n$, $m..n$ denotes the empty set.

A structured constant within a constant list may be defined either by a nested constant list or by a constant of the same type as the component it is associated with.

```
Examples: const stack_limit = 10
          const title = 'Quarterly Report'
          const epsilon = 1E-20
          const basic_colour = red
          const prime_# = array (1..9) of index
                                , (2,3,5,7,11,13,17,19,23:)
          const first_day = date (:19, Sep, 1953:) (*B-2.3.2.*)
          const heat_stroke = disease (:vertigo,dry_skin,
                                       high_pulse:) (*B-2.3.3.*)
          const influenza = disease (:headache,nausea,fever:)
          type sample_record = record
            a,b:      -500..+500
            name:      string(30)
            birth:     date (*B-2.3.2.*)
            #_childs:  0..20
            p:         ->childrec
            xy:         array(1..3,0..1) of boolean
            s:         disease (*B-2.3.3.*)
            r: record
              m:       fix(0..1E6,2)
              n:       float(-1E12..+1E12,7,0.00001)
            end record
          end record
          const init = sample_record
                    (: -50,37,'Harry Lee',(:8,May,1949:),0,nil,
                    (:false,false,true,false,true,true:),
                    heat_stroke,(:153.73,1.382E-3:):)
```

Pascal's syntax of constant declarations has been expanded in a similar way as in Euclid [Lampson et al., 77].

E-3.2. Denotation of constants

A denotation of a constant designates either an entire constant, a substring of a string constant, or a component of a structured constant.

constant ::= entire_const | substr_const | comp_const

An entire constant is denoted either by a literal constant or, if it has been declared in a constant declaration or a declaration of an enumerated type, by its identifier.

entire_const ::= literal_const | ["+"|"-"] constant_id
 literal_const ::= integer_const | real_const | litstring | "nil"

Literal constants have been defined together with their types in B-2.2. and B-2.4. A constant identifier denoting an entire constant of root type integer or real may be optionally preceded by a sign.

A substring of a string constant is denoted by the constant followed by a substring selector specifying the substring.

substr_const ::= constant "<:" expression ["," expression] ">:"

The value of both expressions must be compatible with type integer. Let s be the string denoted by the constant and k be the length of s . Then $s<:m,n:>$ denotes the substring of length n starting at position m , where the first character of s is at position 1. If n is not specified, it is assumed to be $k-m+1$. If all or part of the substring specified lies outside s or if $n < 0$ or if $m < 1$, the program is illegal. If $n = 0$ (either explicitly or by assumption), the substring specifies a valid nullstring for all $m \geq 1$.

A component of a structured constant is denoted by the constant followed by a selector specifying the component. The form of the selector depends on the structuring type of the constant. If the component's root type is integer or real, it may optionally be preceded by a sign.

comp_const ::= ["+"|"-"] (indexed_const | field_const)

An indexed constant is a component of an n-dimensional array and is denoted by the constant followed by n index expressions.

```
indexed_const ::= constant "(" expression {"," expression} ")"
```

The value of the n-th index expression must be compatible with the n-th index type declared in the definition of the array type.

A field constant is a component of a record constant and is denoted by the record constant followed by the field identifier of the component.

```
field_const ::= constant "." field_id
```

```
Examples: stack_limit
          -epsilon
          heat_stroke
          title<:a,2:>
          title<:b+3:>
          'abcde'<:code,1:>
          prime_#(x)
          first_day.Month
          sample_record.r.n
          sample_record.xy(tfunc(x),0)
```

E-3.3. Variable declarations

A variable declaration consists of a list of identifiers denoting the new variables, followed by their type. A bind declaration specifies that the variable identifier is to be bound to an already existing variable rather than to a newly created one. Consequently, the bound variable has the same type as the variable it is bound to by the variable binding declaration. The variable declaration must textually precede any use of the variable.

```
variable_dcl ::= "var" variable_id {"," variable_id} ":"
               type_def |
               "bind" variable_id "to" variable
```

```
Examples: var result: boolean
          var first_name,second_name: string(20)
          var matrix: array (1..100,1..100) of 0..10000
          var a,b,c,d: colour
```

```

var character: set of properties
var p,q,r: record
    angle: float(0..360,7,0.0000001)
    dist: fix(0..1E4,2)
end record
var p1,p2: -> info_rec
bind a to x
bind c to p->elt(j+3)
bind s to cust_rec.cond.rebate

```

Binding will be discussed in B-3.5.

Remember that a variable may not be declared to be of type integer (B-2.2.1.2.), but only of some subrange thereof.

B-3.4. Denotation of variables

A denotation of a variable either designates an entire variable, a substring of a string variable, a component of a structured variable, or a variable referenced by a pointer.

```

variable      ::= entire_var | substr_var | comp_var
                  referenced_var

```

An entire variable is denoted by its identifier. It can be a simple variable, a structured variable, or a pointer variable.

```
entire_var     ::= variable_id

```

A substring of a string variable is denoted by the variable followed by a substring selector specifying the substring.

```
substr_var     ::= variable "<:" expression [ "," expression ] ">:"

```

The value of both expressions must be compatible with type integer. Let s be the string denoted by the variable and k be the actual length of s . Then $s<:m,n:>$ denotes the substring of length n starting at position m , where the first character of s is at position 1. If n is not specified, it is assumed to be $k-m+1$. If all or part of the substring specified lies outside s or if $n < 0$ or if $m < 1$, the program is illegal. If $n=0$ (either explicitly or by assumption), the substring specifies a valid nullstring for all $m \geq 1$.

A component of a structured variable is denoted by the variable followed by a selector specifying the component. The form of the selector depends on the structuring type of the variable.

`comp_var ::= indexed_var | field_var`

An indexed variable is a component of an n-dimensional array variable and is denoted by the variable followed by n index expressions.

`indexed_var ::= variable "(" expression {"," expression} ")"`

The value of the n-th index expression must be compatible with the n-th index type declared in the definition of the array type.

A field variable is a component of a record variable and is denoted by the record variable followed by the field identifier of the component.

`field_var ::= variable "." field_id`

A referenced variable is denoted by the pointer variable denoting its address followed by the pointer symbol "->". The variable is of the pointer's associated type.

`referencd_var ::= variable "->"`

Examples: alpha

```
first_name<:x,y:>
name<:15:>
value_matrix x(a,b)
transaction.branch
first_element->
first_element->.value
family->.father->.child(n).birthdate
```

B-3.5. Binding

An identifier is bound to a variable when it appears

- as a var parameter in a procedure or function declaration (B-7., B-8.)
- in a variable binding declaration (B-3.3.)

A variable to which an identifier is bound is said to be renamed.

The scope of binding is the scope of the block containing the binding declaration, and within this scope the identifier denotes the renamed variable. That is, the identifier has the same type as the renamed variable and also refers to the same storage location. If the renamed variable is an indexed variable, the index is evaluated when binding occurs; if it is a referenced variable, the address is evaluated when binding occurs. A variable cannot be bound to a substring of a string variable.

A component of a packed structure may not be renamed.

If a record variable is to be used a number of times in field designators, it is often convenient to bind a short identifier to it.

```
Example:  begin example
           bind t to trec(n).tpart
           if t.no < t.min then
             t.ord := true
           else
             t.no := t.no - order
           end if
         end begin example
```

is equivalent to

```
if trec(n).tpart.no < trec(n).tpart.min then
  trec(n).tpart.ord := true
else
  trec(n).tpart.no := trec(n).tpart.no - order
end if
```

This usage of binding replaces Pascal's "with" statement.

The idea of binding and some of the wording of this chapter is copied from the Euclid Report [Lampson et al., 77]. Binding can simplify programming, lead to more self-documenting programs, and save execution time.

B-4. EXPRESSIONS

=====

Expressions are constructs denoting rules of computation for obtaining values of variables and generating new values by the application of operators.

The rules of composition specify operator precedences according to eight classes of operators:

- | | |
|-----------------------------|--------------------|
| 1. sign operators (monadic) | highest precedence |
| 2. exponentiation operator | . |
| 3. multiplying operators | . |
| 4. adding operators | . |
| 5. relational operators | . |
| 6. "not" operator | . |
| 7. "and" operator | . |
| 8. "or" operator | lowest precedence |

Sequences of operators of the same precedence are executed from left to right. An expression is always evaluated in its entirety, even if the evaluation of some parts of it may be redundant. The rules of precedence are reflected by the following syntax:

```

simple_factor ::= variable | constant | function_des |
              "(" expression ")" | value_constr |
              sign_op simple_factor
factor       ::= simple_factor | factor "*" simple_factor
term         ::= factor | term mult_op factor
sum          ::= term | sum add_op term
relation     ::= sum | sum rel_op sum
negation     ::= relation | "not" relation
conjunction  ::= negation | conjunction "and" negation
expression  ::= conjunction | expression "or" conjunction

```

In opposition to Pascal, PISA gives the arithmetic and relational operators precedence over the boolean operators. This conforms to a mathematical standard and to the the operator precedences of most programming languages.

E-4.1. Value constructors

A value of a structured type may be built up from values of its component types by a value constructor. A value constructor consists of a type identifier followed by a component list.

```
value_constr ::= type_id comp_list
comp_list   ::= "(" [comp_val {"," comp_val}] ")"
comp_val    ::= expression [".." expression] | comp_list
```

The values of the expressions in the component list become the values of the components of the structured type. The type of the expressions must be compatible with the type of the component they are associated with. A component value of the form $m..n$ denotes a set of elements and is only allowed in a component list for a set type.

For an array type, the order of the values in the component list is the linearized order of the array components. The linearized order of a one-dimensional array is obtained by selecting all its components in ascending order of their index values. The linearized order of an n -dimensional array is produced by rewriting it as n nested one-dimensional arrays and recursively applying the procedure for one-dimensional arrays (row-major).

For a record type, the order of the values in the component list is the order in which the components appear in the record type definition. The number of values in the component list must be equal to the number of record components. A value constructor may not be used for record types with a variant part.

A set type contains all elements enumerated in the component list. The values in the component list must be of the set's base type. An empty component list denotes an empty set. A component value of the form $m..n$ denotes the set of all elements i in the subrange $m..n$ of the base type such that $m \leq i \leq n$. If $m > n$, $m..n$ denotes the empty set.

A structured value within a component list may be defined either by a nested component list or by an expression of the same type as the component it is associated with.

Examples: date(:15,actual_month,actual_year+1900:)
 sample_record(:m,n,'John',(:day,month,year:),0,cp,
 (:b1,b2,b3,b4,b5,b6:),illness,(:x1,x2-100:):)

The type "date" has been defined in an example in B-2.3.2., the type "sample_record" in B-3.1.

E-4.2. Operators

Each operator defines the type of its operands. The value of the actual operands must be compatible with this type (cf. B-2.5.).

E-4.2.1. Sign operators

The sign operators are monadic (prefix) operators.

sign_op ::= "+" | "-"

<u>oper</u>	<u>operation</u>	<u>type of operand</u>	<u>type of result</u>
+	identity	integer, real	same as operand
-	sign inversion	integer, real	same as operand

E-4.2.2. Exponentiation operator

<u>oper</u>	<u>operation</u>	<u>type of operands</u>	<u>type of result</u>
**	exponentiation	real	internal float

The result of an exponentiation is always of type internal float, independent of the types of its operands. The type internal float is a floating point type defined by "float(-a..+a,m,z)". -a..+a represents the largest range, m the largest number of decimal mantissa digits, and z the smallest nonzero value accepted in explicit float type definitions (B-2.2.2.). A value of type internal float is usually represented by a normalized

floating point value as accurate as the underlying hardware allows.

When the result of an exponentiation is a complex value, the program is illegal.

B-4.2.3. Multiplying operators

mult_op ::= "*" | "/" | "div" | "mod"

<u>oper</u>	<u>operation</u>	<u>type of operands</u>	<u>type of result</u>
*	multiplication set intersection	integer, real any set type t	see below t
/	division	real	internal float
div	division with truncation	integer	integer
mod	modulo division	integer	integer

If both operands of a multiplication are of root type integer, the result is of type integer too. If at least one of the operands is of root type real, the result is of type internal float.

The "div" operator uses integer division and normally requires considerably less execution time than the "/" operator. Truncation is toward zero, so that $-(a \text{ div } b) = -a \text{ div } b$. Also $a \text{ div } -b = -a \text{ div } b$. The "mod" operator is defined by: $a \text{ mod } b = a - (a \text{ div } b) * b$. The right operand of "/", "div" and "mod" must be nonzero.

The set intersection $a * b$ denotes the set of all elements of a that are also elements of b.

B-4.2.5. Relational operators

```

rel_op      ::= ["rep"] comp_op | "in"
ccmp_op     ::= "=" | "<>" | "<" | ">" | "<=" | ">="

```

<u>oper</u>	<u>operation</u>	<u>type_of_operands</u>	<u>type_of_result</u>
= <>	equality / inequality	any simple type	boolean
	equality / inequality	any set type t	boolean
	equality / inequality	any pointer type	boolean
< >	less / greater	any simple type	boolean
<=	less or equal	any simple type	boolean
	set inclusion	any set type t	boolean
>=	greater or equal	any simple type	boolean
	set inclusion	any set type t	boolean
in	set membership	right: any set type t left: t's base type	boolean

The values of the two operands in a relation test other than set membership must be compatible. If type adaptation is necessary, the following rules apply before the relation test is executed:

- Subranges are adapted to their root types.
- From two real values in a relation test, the one with the less accurate precision is adapted to the type of the other. A floating point value is always considered to be preciser than a fixed point value.
- The shorter string is adapted to the type of the longer, where the length is the actual length of the string (cf. B-2.2.3.).

If conversion of one operand is required before the relation test is executed, the following rules apply:

- The integer operand is converted to the type of the real operand.

- The char operand is converted to the type of the string operand.

Relation tests other than "=" and "<>" are implementation dependent when applied to string or char types.

The result of a set inclusion test $a \leq b$ or $b \geq a$ is true when all elements of a are also elements of b , false otherwise.

The result of a set membership test is true when the first operand is an element of the second, false otherwise.

Relation tests involving real types are defined such that the two real values a and b are considered to be equal (=) if the relation

$$|r(a) - r(b)| < f$$

holds, otherwise they are considered to be unequal (<>). The so called "fuzz value" f is equal to the smallest nonzero value z accepted in float type definitions (B-2.2.2.). The function r rounds the value of its actual parameter to the accuracy given by the actual parameter's type:

- a fix value is rounded to as many decimal fractional digits as given by the actual parameter's type.
- a float value is rounded to as many decimal mantissa digits as given by the actual parameter's type.

If the actual parameter is a constant, no rounding is performed, i.e., the function r returns the value of the constant. If a type's internal representation maintains exactly as many decimal digits as the type definition specifies, the function r is redundant for this type.

The definition of equality of two real values implies the following relation tests on the real values a and b :

$a < b$	is true iff	$r(a) - r(b) \leq -f$
$a > b$	is true iff	$r(a) - r(b) \geq f$
$a \leq b$	is true iff	$r(a) - r(b) < f$
$a \geq b$	is true iff	$r(a) - r(b) > -f$

If the relational operator specifying a comparison of two real values is preceded by the symbol "rep", the two real values are compared as they are internally represented (i.e., without any rounding as described above). A relation test " $x \text{ rep} = y$ " is read as "the internally represented value of x equal to the internally

represented value of y ", the other relation tests with the "rep" option accordingly. Relation tests with the "rep" option are implementation dependent. Their advantage over the relation tests on rounded values is that they can be executed much faster on most implementations. The symbol "rep" may only be used in relation tests on real types.

F-4.2.6. Boolean operators

<u>OPER</u>	<u>operation</u> -----	<u>type of operands</u>	<u>type of result</u>
not	logical negation	boolean	boolean
and	logical AND	boolean	boolean
or	logical OR	boolean	boolean

"not" is a monadic (prefix) operator and has precedence over "and" which has precedence over "or".

B-4.3. Function designators

A function designator specifies the evaluation of a function. It consists of the identifier designating the function and, optionally, a list of actual parameters. The actual parameters are substituted in place of the corresponding formal parameters declared in the function declaration. The correspondence is established by the order of the parameters in the lists of actual and formal parameters, respectively. There exist three kinds of parameters:

1. In the case of a value parameter, the actual parameter must be an expression, of which a variable is a simple case. The value of the expression must be compatible with the type of the formal parameter. A structured type cannot be a value parameter.
2. In the case of a variable parameter, the actual parameter must be a variable of the same type as the formal parameter. Components of a packed structured type may not appear as variable parameters. When the designated

function assigns a value to a formal variable parameter, it is assigned to the actual parameter.

3. In the case of an area parameter, the actual parameter can be a variable of any type or a formal area parameter. Components of a packed structured type may not appear as area parameter. Area parameters are used for data exchange between a PISA program and the run-time system and will be explained in B-11.2.

```
function_des ::= function_id [act_parmlist]
act_parmlist ::= "(" actual_parm {"," actual_parm} ")"
actual_parm  ::= expression | variable
```

```
Examples: ord(char,'X')
          pi
          GCD(147,k)
          time(x,y(3))
```

E-5. BLOCKS AND SCOPES OF IDENTIFIERS

A FISA program is composed of one or more blocks of code, briefly called blocks. There are nine kinds of blocks according to the constructs which define these blocks:

procedure blocks	established by	procedure declarations
function blocks	"	" function declarations
begin blocks	"	" compound statements
then blocks	"	" if statements
else blocks	"	" if statements
when blocks	"	" case statements
otherwise blocks	"	" case statements
action blocks	"	" decision table statements
loop blocks	"	" repetitive statements

Independent of its kind, every block has exactly the same basic structure and the same two operations defined on it: block activation and block termination. The blocks merely differ in the way their textual range is defined and in the method of their activation. The range of blocks and the conditions for their activation will be defined in subsequent chapters.

This chapter defines the basic structure of blocks as well as block activation and block termination.

E-5.1. Blocks

A block consists of a declaration part followed by a statement part. Variables, constants, types, procedures and functions are declared in the declaration part. The statement part consists of zero or more statements.

```

block      ::= decl_part stmt_part
decl_part  ::= {declaration}
declaration ::= variable_dcl | constant_dcl | type_dcl |
                    procedure_dcl | function_dcl
stmt_part  ::= {statement}

```

Procedure declarations, function declarations, and structured statements contained within a block constitute other blocks in turn. These blocks are called nested blocks and are part of the surrounding block.

The operations defined on a block are activation and termination. A block is always activated by the construct defining the block and always terminates itself.

Block activation causes binding of the variables declared to bind for the block (B-3.3.). The values of the other variables declared in the block are undefined after its activation. After the binding, the statements of the block are executed sequentially. Block activation does not allocate data space for the objects declared in the block. The data space is allocated and freed jointly for the whole procedure or function (B-7.). The compiler is expected to overlay the variables of the blocks according to their hierarchical structure within the procedure or function.

Block termination causes discontinuing the execution of its statement part and unbinding of all variables bound in this block. Block termination occurs:

- after the last statement of the block has been executed, or
- by executing an exit statement for the block or any surrounding block, or
- when control leaves the block by a goto statement designating a label declared in a surrounding block.

A block that can be designated by an identifier is called a named block. The name of the block is declared in its defining construct:

- a procedure or function block inherits the name of the procedure or function, resp.
- a begin block is named by the block identifier in the compound statement (B-6.3.1.).
- a loop block is named by the block identifier in the repetitive statement (B-6.3.3.).

Other blocks cannot be named. Only named blocks may be terminated by an exit statement.

B-5.2. Scope of identifiers

Identifiers declared in a block are known within this block including its nested (enclosed) blocks, collectively called the scope of the identifiers. As an exception, the scope of label identifiers does not include the scope of nested procedure or function blocks.

An identifier may not denote more than one object in the proper text of a block. The proper text of a block is that part of its text that is not part of nested blocks. If several objects have the same name within the actual scope, the corresponding identifier denotes the object that has been declared last.

If an identifier *x* used within a block *b* refers to an object declared in a block enclosing *b*, *x* may not be redeclared in the proper text of *b* (see above for explanation of "proper text"). Hence, the following PISA program segments are all illegal because line 5 contains an identifier referring to an object declared in *b1* and line 6 illegally redeclares this identifier.

1	begin b1	1	begin b1
2	const c = 7	2	var p1: ->elt
3	print(c)	3	new(p1)
4	begin b2	4	begin b2
5	const d = c	5	bind x to p1->
6	const c = 0	6	var p1: ->struc
7	end begin b2	7	end begin b2
8	end begin b1	8	end begin b1

1	begin b1
2	*> L:
3	print('a')
4	begin b2
5	goto L
6	*> L:
7	end begin b2
8	end begin b1

B-6. STATEMENTS
=====

Statements denote algorithmic actions and are said to be executable. They may be prefixed by one or more label declarations.

```
statement ::= {label_dcl} (simple_stmt | struc_stmt)
```

B-6.1. Label declarations

Labels are identifiers that can be referenced by goto statements. They are defined in a label declaration preceding a statement.

```
label_dcl ::= ">" label_id {"," label_id} ":"
```

A label identifier is known within that part of the current scope that is not part of nested procedures or functions. A label identifier need not be declared before it is referenced.

Pascal's integer labels do not support self documentation of programs in a satisfactory way. The above syntax introduces label identifiers without enforcing a lookahead parse of the program text as would be necessary for a PL/I-like label syntax.

B-6.2. Simple statements

A simple statement is a statement of which no part establishes a (nested) block.

```
simple_stmt ::= assign_stmt | proc_stmt | goto_stmt |
            exit_stmt | suspend_stmt
```

B-6.2.1. Assignment statements

The assignment statement serves to set the value of a variable or function to the value specified by an expression.

```
assign_stmt ::= (variable | function_id) "=" expression
```

The type of the expression must be compatible with the type of the variable or function. If the expression is a value constructor, the structured value is evaluated completely before it is assigned to the target variable.

```
Examples:  i := i+j
           going_to_work := week(:mon..fri:)-days_off
           therapy_1 := fever in illness and age < 5
           p->.x(j) := cond.rate(j)
           p-> := first->
           name := first_name<:1,1:> + ' ' + last_name
           name<:6,4:> := 'abcd'
```

B-6.2.2. Procedure statements

A procedure statement serves to activate the procedure denoted by the procedure identifier. The procedure statement may contain a list of actual parameters which are substituted in place of the corresponding formal parameters declared in the procedure declaration. The correspondence is established by the order of the parameters in the lists of actual and formal parameters. There exist three kinds of parameters:

1. In the case of a value parameter, the actual parameter must be an expression, of which a variable is a simple case. The value of the expression must be compatible with the type of the formal parameter. A structured type cannot be a value parameter.
2. In the case of a variable parameter, the actual parameter must be a variable of the same type as the formal parameter. Components of a packed structured type may not appear as variable parameters. When the designated procedure assigns a value to a formal variable parameter, it is assigned to the actual parameter.

3. In the case of an area parameter, the actual parameter can be a variable of any type or a formal area parameter. Components of a packed structured type may not appear as area parameter. Area parameters serve for data exchange between the PISA program and the run-time system and will be explained in B-11.2.

```
proc_stmt      ::= procedure_id [act_parmlist]
act_parmlist   ::= "(" actual_parm {"," actual_parm} ")"
actual_parm    ::= expression | variable
```

```
Examples:  next
           open(inp_file, 'input')
           read(inp_file, rec_trans)
           GCD(x(i+3), a.w)
```

B-6.2.3. Goto statements

A goto statement is used to indicate that further processing should continue at another part of the program text, namely at the place of the label declaration. All blocks left by the execution of a goto statement are terminated (cf. B-5.1).

```
goto_stmt ::= "goto" label id
```

Notice that the definition of the scope of label identifiers (B-5.2.) allows neither jumping across a procedure or function border nor jumping into an enclosed block.

```
Example:      :
              :
      *> L1:  :
              :
              goto L2
              :
              goto L1
              :
      *> L2:  :
              :
              :
```

The arbitrary use of `gotos` can lead to very badly structured programs. Therefore, it might seem preferable to exclude `goto`

statements from PISA's control structure primitives. The main reason for still defining goto statements (and label declarations) lies in PISA's potential use as language front end processors' target code (A-1.3.): Although feasible, it is extremely difficult to translate certain nonprocedural languages to a goto-less PISA program. If the control structure of the program generated by a front end processor is transparent to the user, there are no objections to goto statements.

B-6.2.4. Exit statements

The exit statement indicates that all enclosing blocks up to and including the block designated by the block identifier are to be terminated immediately. The designated block must be contained in the enclosing procedure or function. If the exit statement designates the enclosing procedure or function block, it serves to return from it. Only named blocks can be designated by an exit statement.

```
exit_stmt      ::= "exit" block_id
```

B-6.2.5. Suspend statements

A suspend statement indicates suspension of the executing coroutine procedure or coroutine function and return of control to the construct that invoked this coroutine (procedure statement or function designator). Coroutine mechanisms will be introduced in B-7.1. If the suspend statement causes suspension of the main procedure (B-9.1.), it is ignored.

```
suspend_stmt  ::= "suspend"
```

B-6.3. Structured statements

Structured statements are constructs defining (nested) blocks which have to be activated either unconditionally (compound statement), conditionally (conditional statement), or repeatedly (repetitive statement).

```
struc_stmt    ::= compound_stmt | conditnl_stmt | repetit_stmt
```

E-6.3.1. Compound statements

A compound statement specifies that its block is to be activated unconditionally when control reaches the compound statement.

```
compound_stmt ::= "begin" block_id
                begin_block
                "end" "begin" block_id
begin_block   ::= block
```

The "begin" and "end" constructs of the compound statement define the limits of the block's textual range. The begin block is named by the identifier following the symbol "begin". This identifier is called a block identifier and must be repeated in the "end" construct closing the begin block. The block identifier belongs to the scope of the block it denotes.

```
Example: begin check
          var x: fix(0..1000,2)
          bird a to tab(i+n)
          bind b to rate(year,code)
          if b=0 then
            print ('no rate available')
            exit check
          end if
          x := a/(100+b) * b
          if x<1.5 then x:=1.5 end if
          print('rate=',x)
        end begin check
```

E-6.3.2. Conditional statements

A conditional statement activates a (possibly empty) set of its blocks depending on a defined condition.

```
conditnl_stmt ::= if_stmt | case_stmt | detab_stmt
```

E-6.3.2.1. If statements

The if statement specifies that the then block is to be activated only if a certain condition (denoted by a expression of type boolean) is true. If it is false, then either no block is to be activated, or the else block is to be activated.

```

if_stmt      ::= "if" expression
                "then" then_block
                [ "else" else_block ]
                "end" "if"
then_block   ::= block
else_block   ::= block

```

The expression after the symbol "if" must be of type boolean.

The textual range of a then block is closed by the "else" construct or the end of the if statement, the one of an else block by the end of the if statement.

```

Example:  if a<b then
           x := 1
           c := false
        else
           if a=b then
               if b in critical_range then
                   x := 2
                   a := 1
               else
                   c := true
               end if
           end if
           print(a)
        end if

```

Notice that this syntax does not introduce the syntactically ambiguous if-then-else constructs that Pascal does (cf. Pascal report 9.2.2.1.).

B-6.3.2.2. Case statements

The case statement consists of an expression (the selector) and a list of blocks, each constituted by a when construct which contains a set of constants of the selector's type. The selector must be of a discrete type other than integer and specifies that the one block is to be activated whose when list contains the current value of the selector.

The when lists must be disjoint. Furthermore, the union of the lists must be equal to the set of values defined for the selector's type unless there is an otherwise construct. An otherwise construct comprises all selector values in the set difference S between the set of values defined for the selector

and the union of the when lists, i.e., it lumps all selector values not mentioned explicitly. If there is an otherwise construct and S is empty, the program is illegal.

```

case_stmt      ::= "case" expression
                  {when_constr}
                  [otherwise_constr]
                  "end" "case"

when_constr    ::= "when" when_list ":" when_block
otherwise_constr ::= "otherwise" ":" otherwise_block
when_list      ::= when_item {"," when_item}
when_item      ::= entire_const [".." entire_const]
when_block     ::= block
otherwise_block ::= block

```

A when item $m..n$ denotes all elements i in the subrange $m..n$ of the selector's type such that $m \leq i \leq n$.

The textual range of a when block is closed by the succeeding when construct, by the otherwise construct, or by the end of the case statement, whichever occurs first. An otherwise block is closed by the end of the case statement.

```

Examples: case type_of_car
           when Ford,Mercury:
             bind var x to p(5)
             x := x + 1
             rebate := 10
             wait_for_delivery := 15
           when Chevrolet,Volvo,Pontiac:
             rebate := 15
             wait_for_delivery := 30
           when Mercedes:
             rebate := 0
             wait_for_delivery := 300
           otherwise:
             rebate := 7.5
             wait_for_delivery := on_stock
         end case

case operator
  when plus:   x := x+y
  when minus:  x := x-y
  when times:  x := x*y
end case

```

P-6.3.2.3. Decision table statements

The if statement serves to conditionally activate one block or, when an else clause is included, to activate one out of two blocks depending on the value of one boolean expression. The case statement, as a generalization of the if statement, allows the activation of one out of an arbitrary number of blocks depending on the value of one expression of any discrete type. The decision table statement specifies the activation of several action blocks depending on the values of an arbitrary number of boolean expressions (limited entry decision table).

A decision table statement consists of the statement brackets "detab" and "end detab", a condition part, and an action part.

```
detab_stmt      ::= "detab"
                    cond_part
                    action_part
                    "end" "detab"
cond_part       ::= condition {condition}
condition       ::= "cond" expression cond_entry
action_part     ::= action {action}
action          ::= "action" action_entry
                    action_block
action_block    ::= block
```

The expression after "cond" must be of type boolean. Condition and action entries have been introduced in the definition of FISA's lexical structure (B-1.3.).

```
Example: detab
        cond c1          "-YYYNY"
        cond c2          "-NNY-Y"
        cond c3          "-NY---"
        cond c4          "-NNNY"
        action           "-XX---"
            (*actionblock 1*)
        action           "-X-XX-"
            (*actionblock 2*)
        action           "X--X-X"
            (*actionblock 3*)
    end detab
```

c1, c2, c3, and c4 represent arbitrary expressions of type boolean. Each comment in the action part stands for an action block.

The following paragraphs define the semantics of the decision table statement. The PISA program fragments therein are used as

a metalanguage to support the definition; they are not necessarily the actual implementation of decision table statements.

Each decision table defines 32 decision rules or just rules, numbered from rule 1 to rule 32. Rule 1 is also called the else rule. During evaluation of a decision table's condition part, it turns out that some of the 32 rules hold while the others do not hold. A rule that holds is known as an active rule; a rule that does not hold is a passive rule. The state of the 32 decision rules is represented by a so called rule mask which is the set of the active rules.

```
type rule = (R1,R2,R3,R4...R29,R30,R31,R32)
const else_rule = R1
type rules = set of rule
var rule_mask: rules
```

Before the condition part of the decision table statement is evaluated, all rules are activated except the else rule.

```
rule_mask := rules(:R2..R32:)
```

The conditions in the condition part are evaluated in the sequence they are written. The rule tokens (B-1.3.) in condition entries designate the rules 1 to 32 according to their ordering within the condition entry (the i-th token designates rule i). If a condition entry contains fewer than 32 rule tokens, the missing tokens are considered as "-" tokens. A rule token "Y" or "y" (a YES token) indicates that the designated rule should remain active if and only if it was active before this condition was evaluated and if the value of the boolean expression of the condition is true. A rule token "N" or "n" (a NO token) indicates that the designated rule should remain active if and only if it was active before this condition was evaluated and if the value of the boolean expression of the condition is false. A rule token "-" specifies that the designated rule is not affected by this condition. A "-" token is therefore also known as an "irrelevant" or "don't care" token. According to the definition of condition entries (B-1.3.), the else rule (rule 1) cannot be designated by a YES or NO token.

When control reaches a condition and no YES or NO token of this condition designates an active rule, the boolean expression in this condition is not evaluated. The following PISA program fragment defines the evaluation of a condition and is based on the second condition in the example on the preceding page.


```

begin evaluate_cond_2
  const yes_mask = rules(:R4,R6:)
  const no_mask  = rules(:R2,P3:)
  if (yes_mask+no_mask)*rule_mask <> rules(:) then
    if c2 then
      rule_mask := rule_mask-no_mask
    else
      rule_mask := rule_mask-yes_mask
    end if
  end if
end begin evaluate_cond_2

```

If no rules remain active after all conditions have been evaluated, the else rule is activated before the action part of the decision table statement gains control.

```

if rule_mask = rules(:) then
  rule_mask := rules(:else_rule:)
end if

```

The actions are evaluated in the order they are written. A block in an action is activated only if at least one of the rules designated by a rule token "X" or "x" is active. The relation between rule tokens in action entries and the rules they designate is the same as defined for condition entries. The following PISA program fragment describes the evaluation of an action entry and is based on the second action in the example given two pages before.

```

if rule_mask*rules(:R2,R4,R5:) <> rules(:) then
  (*actionblock 2*)
end if

```

The textual range of an action block is closed by the succeeding action or the end of the decision table statement, whichever occurs first.

```

Example:  detab
          cond item_on_stock           "yyynn"
          cond number_on_stock >= number_ordered "ynn--"
          cond customer = vip          "--ynyn"

          action                       "-x---"
            print('items on stock')
          action                       "----xx"
            print('items not on stock')
          action                       "--xx--"
            print('not enough items on stock')
          action                       "-xxxx-"
            print('accept order')
          action                       "-----x"
            print('reject order')
          action                       "--x-x-"
            print('buy missing items from')
            print('supplier and send')
            print('directly to customer')
          action                       "-x-x--"
            print('ship order from stock')
          action                       "----x--"
            print('rest will follow soon')
end detab

```

The decision table statement as defined in this chapter is known as a procedural limited entry decision table based on the rule mask technique, first introduced by Thurner [Thurner,71], [Thurner,73]. A widely referenced publication on theory and practice of decision tables is [Pollack et al.,71].

B-6.3.3. Repetitive statements (loop statements)

A repetitive statement specifies that its loop block is to be activated repeatedly.

```

repetit_stmt ::= "loop" block_id [for_constr | while_constr]
              loop_block
              "end" "loop" block_id
for_constr   ::= "for" control_var "!=" initial_value
              ("to"|"downto") final_value
while_constr ::= "while" expression
control_var  ::= identifier
initial_value ::= expression
final_value  ::= expression
loop_block   ::= block

```

The repetitive statement denotes the textual range of the loop block. The loop block is named by the identifier following the symbol "loop". This identifier must be included in the "end" construct closing the loop block. The block identifier belongs to the scope of the block it denotes.

A loop statement without a for or while construct causes repeated activation of the loop block until the block is terminated by an exit statement.

A for construct is used if the number of repetitions is known before the repetition starts. The semantics of the for construct is defined by the following translations, where the type of i is represented by t . i must be an entire variable of a discrete type other than integer.

```
loop x for i:=m to n
  (* loop block *)
end loop x
```

translates to

```
c1 := m
c2 := n
i := c1
if i<=c2 then
  loop x
    begin loop_block
      (* loop_block *)
    end begin loop_block
    if i>=c2 then
      exit x
    end if
    i := succ(t,i)
  end loop x
end if
```

```
loop x for i:=m downto n
  (* loop block *)
end loop x
```

translates to

```
c1 := m
c2 := n
i := c1
if i>=c2 then
  loop x
    begin loop_block
      (* loop_block *)
    end begin loop_block
    if i<=c2 then
      exit x
    end if
    i := pred(t,i)
  end loop x
end if
```

The identifiers $c1$, $c2$, and `loop_block` are used as "meta identifiers" above to denote compiler generated objects. They cannot be denoted in a PISA program. The type of $c1$ and $c2$ is t (the type of i).

If any of the statements which are the translation of the loop statement with a for construct is invalid, then the loop statement is invalid.

The while construct specifies the repeated activation of the loop block as long as the boolean expression following the symbol "while" is true:

```
loop x while b
  (* loop block *)
end loop x
```

translates to

```
loop x
  if b then
    (* loop block *)
  else exit x end if
end loop x
```

Examples:

```
loop next_command
  print('enter command'); input c
  if len(c)=1 then exit next_command end if
  print('invalid command syntax')
end loop next_command

i := 0
loop count for symptom := headache to high_pulse
  if symptom in illness then i:=i+1 end if
end loop count
if i>6 then print('poor fellow') end if

p := first_elt
loop print_list while p<>nil
  print(p->.text)
  p := p->.next
end loop print_list
```

The loop statement as defined above has been inspired by the loop and for statements of Euclid [Lampson et al.,77]. In combination with the exit statement, it replaces Pascal's while, repeat, and for statements. A detailed discussion of the capabilities of repetitive and exit statements is given in [Peterson et al.,73].

B-7. PROCEDURE DECLARATIONS

=====

Procedure declarations serve to define a block of code and to associate an identifier with it, so that it can be executed by procedure statements. A procedure must be declared before it is referenced in a procedure statement.

```

procedure_dcl ::= proc_def | external_proc | forward_proc
proc_def      ::= ["coroutine"] "procedure" procedure_id
                  [form_parmlist]
                  proc_block
                  "end" "procedure" procedure_id
proc_block    ::= block
form_parmlist ::= "(" formal_parm {"," formal_parm} ")"
formal_parm   ::= ["var"] identifier ":" type_def |
                  "area" identifier

```

B-7.1. Procedure definitions

The procedure definition specifies the formal parameters of the procedure (if any). When the procedure is called, the actual parameters of the procedure statement are substituted for the formal parameters.

The parameters are either value parameters, variable parameters, or area parameters.

A formal value parameter represents a local variable of the procedure. The value of the actual parameter is assigned to it as an initial value. A value parameter may not be of a structured type. A formal parameter preceded neither by the symbol "var" nor by the symbol "area" is a value parameter.

A formal variable parameter is bound to its actual parameter. It is preceded by the symbol "var" in the formal parameter list.

A formal area parameter is bound to its actual parameter but has no type and cannot be denoted in a PISA program except as actual area parameter. It is preceded by the symbol "area" in the formal parameter list. Area parameters are used for data exchange with the PISA run-time system and will be explained in B-11.2.

A procedure definition prefixed with the symbol "coroutine" introduces a coroutine procedure; otherwise it introduces a simple procedure. A coroutine procedure may not be declared within a simple procedure or simple function. As defined in E-9.1., the main procedure (or program) is implicitly a coroutine procedure.

When a simple procedure is invoked, it is "seized". "Seizing" causes the allocation of data space for all static variables in the procedure and the association of the formal parameters with the actual ones as described above. After the procedure has been seized, the procedure block is activated (B-5.1.).

Termination of a simple procedure's block causes the procedure to be "released" and control to be returned to the invoking procedure statement. "Releasing" deallocates the static variables of the procedure; dynamic variables allocated in the procedure are not freed when the procedure is released.

As long as no suspend statement is executed, a coroutine procedure behaves exactly as a simple procedure: it is seized when it is entered, and it is released when the coroutine procedure block terminates. Execution of a suspend statement causes "suspension" of the coroutine procedure. The suspend statement can be textually part of the coroutine procedure block's statement part, or it can be executed in any simple procedure or simple function (B-8.1.) directly or indirectly called from the coroutine procedure. "Suspension" returns control to the procedure statement that invoked the coroutine procedure yet conserves the state of the coroutine. In other words, the coroutine's actual structure of procedure and function activations is retained.

When a suspended coroutine procedure is invoked anew, the actual parameters are resubstituted for the formal ones as defined above; thereafter, execution continues at the place of the suspend statement that caused the coroutine's previous suspension. Resubstituting the coroutine's formal parameters does not rebind variables and parameters bound to them; they remain bound to the actual parameters they were originally bound to. Similarly, resubstitution does not reinitialize formal value parameters of nested procedures and functions.

If a coroutine is released, all its textually enclosed coroutines that are not yet released are released as well.

Invoking a simple procedure already seized is called recursive invocation. The procedure is seized for each recursive invocation. A coroutine procedure may only be invoked when it is

not yet seized or when it is suspended, recursive coroutine invocation is illegal.

In contrast to a completely symmetric relationship between coroutines as described in [Knuth,68], the coroutines of PISA reflect the usual hierarchical (or master-slave) relationship between procedures. They still denote levels of abstraction, even if some of the levels are implemented as coroutines.

Examples:

```

procedure error(no: 1..200)
  bind errortext to err_tab(no)
  var answer: string(3)
  if no<100 then
    print('***'+errortext)
    exit error
  end if
  print('!!!'+errortext+bell)
  printnr('continue? ')
  input(answer)
  if answer='y' or answer='yes' then exit error end if
  stop('')
end procedure error

coroutine procedure get(area rec, var endfile: boolean)
  var eof:      boolean
  var file_#:  0..1000
  open('file_15','input',file_#)
  loop read_next_record
    read(file_#,rec,eof)
    if eof then exit read_next_record end if
    endfile := false
    suspend
  end loop read_next_record
  close(file_#)
  endfile := true
  suspend
  stop('you tried to read beyond the end of the file')
end procedure get

```

A more complex example of nested coroutines is given in B-8.1.

B-7.2. External procedures

If a procedure designated in a procedure statement is not defined in the actual program, it is called an external procedure. External procedures are stored in compiled form on a storage medium and are linked to the actual program after its compilation.

```
external_proc ::= "procedure" procedure_id "external"
                [ parm_typelist ]
parm_typelist ::= "(" parm_type { "," parm_type } ")"
parm_type     ::= [ "var" ] type_def | "area"
```

A declaration of an external procedure does not declare parameter identifiers; it merely defines the types of the formal parameters (if any). The type definition must exactly correspond to the formal parameter list of the designated procedure.

An implementation may impose restrictions upon the syntax of external procedure identifiers.

The predefined procedures (B-10., B-11.) must not be declared as external procedures since they are virtually defined in a hypothetical block surrounding a PISA program.

Examples:

```
procedure read_key external (string(10), var boolean, area)
procedure check external
```

B-7.3. Forward procedure declarations

If a procedure is used in a procedure statement before it is defined, it must be declared with a forward declaration. The procedure has to be defined later on in the declaration part of the block containing the forward declaration.

```
forward_proc ::= "procedure" procedure_id "forward"
                [ parm_typelist ]
```

The parameter type list has been defined in B-7.2.

A forward declaration does not declare parameter identifiers; it merely defines the types of the formal parameters (if any). The type definition must exactly correspond to the formal parameter list of the procedure definition for this procedure.

Examples:

```
procedure rec_proc forward (boolean, var char, 5..50)
procedure expression forward
```

B-8. FUNCTION DECLARATIONS

=====

Function declarations serve to define parts of the program which compute a value of a simple or pointer type. Functions are activated by the evaluation of a function designator (B-4.3.) which is a constituent of an expression. A function must be declared before it is referenced by a function designator.

```
function_dcl ::= function_def | external_func | forward_func
function_def ::= ["coroutine"] "function" function_id
                [form_paramlist] ":" type_def
                func_block
                "end" "function" function_id
func_block   ::= block
```

The formal parameterlist has been defined in B-7.

B-8.1. Function definitions

A function is essentially a procedure that returns a value denoted by the function designator in an expression. With the obvious textual adaptations, the definitions of B-7.1. apply to functions as well and are not repeated here.

Within a function block, the function can be denoted like a variable (B-3.4.). It represents the value returned by the function. The type of the function is given by the type definition in the function definition which must designate a simple or a pointer type. Before a function is released or suspended, its value must have been set by one or more assignments to the function identifier, otherwise its value is undefined.

Example:

```
(* merge two binary trees by inorder traverse *)

coroutine function merge(root1:->elt, root2:->elt): ->elt

  var t1,t2: ->elt

  procedure traverse(p: ->elt, var p1: ->elt)
    if p = nil then exit traverse end if
    traverse(p->.left,p1)
    p1 := p
    suspend
    traverse(p->.right,p1)
  end procedure traverse

  coroutine procedure tree1
    traverse(root1,t1)
    t1 := nil
  end procedure tree1

  coroutine procedure tree2
    traverse(root2,t2)
    t2 := nil
  end procedure tree2

  tree1; tree2
  loop merge_trees
    detab
      cond t1 = nil                                "--yynnnn"
      cond t2 = nil                                "--ynynn"
      cond t1->.key > t2->.key                      "----yn"
      action                                         "-x-----"
        merge := nil; exit merge
      action                                         "----x-x"
        merge := t1; tree1
      action                                         "---x-x-"
        merge := t2; tree2
    end detab
    suspend
  end loop merge_trees
end function merge
```

E-8.2. External functions

If a function designated in a function designator is not defined in the actual program, it is called an external function. External functions are stored in compiled form on a storage medium and are linked to the actual program after its compilation.

```
external_func ::= "function" function_id "external"
                [ param_typelist ] ":" type_def
```

The parameter type list has been defined in B-7.2.

A declaration of an external function does not declare parameter identifiers; it merely defines the types of the formal parameters (if any). Their types and the type of the function must exactly correspond to the definition of the designated function.

An implementation may impose restrictions upon the syntax of external function identifiers.

The predefined functions (E-10., E-11.) must not be declared as external functions since they are virtually defined in a hypothetical block surrounding a PISA program.

Examples:

```
function matinv external (matrix): matrix
function GCD external (1..10000, 1..10000): 1..10000
```

E-8.3. Forward function declarations

If a function is used in a function designator before it is defined, it must be declared with a forward declaration. The function has to be defined later on in the declaration part of the block containing the forward declaration.

```
forward_func ::= "function" function_id "forward"
                [ param_typelist ] ":" type_def
```

The parameter type list has been defined in B-7.2.

A forward declaration does not declare parameter identifiers; it merely defines the types of the formal parameters (if any). Their types and the type of the function must exactly correspond to the definition of the designated function.

Examples:

```
function rec_function forward (1..100,boolean): ptr  
function nextit forward: angle
```

B-9. PROGRAMS AND EXTERNAL ROUTINES

=====

A PISA program is a procedure invoked by the PISA run-time system. It may contain declarations of external procedures and external functions, collectively called external routines. External routines are compiled separately and are linked to a PISA program after compilation of the program.

A PISA compilation either translates a program or an external routine.

compilation ::= program | ext_routine

B-9.1. Programs

A PISA program is established by a so called main procedure. A main procedure may not be invoked by a procedure statement. It is elected for execution by the PISA run-time system and has no formal parameter list. The main procedure is implicitly a coroutine procedure.

program ::= "main" proc_def

An implementation may impose restrictions upon the syntax of main procedure identifiers.

Example:

```
main procedure testprog
  var i: 1..100
  loop squares for i := 1 to 100
    print (i,'*',i,' = ',i*i)
  end loop squares
end procedure testprog
```

B-9.2. External routines

An external routine is a procedure or function that can be designated by an external procedure declaration (B-7.2.) or an external function declaration (B-8.2.) respectively. An external

routine is a procedure or function definition preceded by the symbol "external".

```
ext_routine    ::= "external" (proc_def | function_def)
```

An implementation may impose restrictions upon the syntax of external procedure and function identifiers.

The type of the formal parameters of an external routine may be defined by a type identifier or a type definition. If a type identifier is used, it must be declared within the declaration part of the procedure or function block and before the parameter is referenced. Similarly, if a type identifier defines the type of an external function, this identifier must be declared in the function block's declaration part and before a value is assigned to the function. These are necessary exceptions from the rule that a type identifier must be declared before it is used since there is no place where it could be declared textually preceding such a usage.

An external routine may be a simple routine or a coroutine. It may contain external procedure and external function declarations in turn.

```
Example: external coroutine function primeno: pos_int
      type pos_int =    0..1000000
      var i,j,k:        pos_int

      primeno := 2
      suspend
      i := 3
      loop odd_numbers while i<999999
        begin is_it_prime
          k := trunc(sqrt(i)); j := 3
          loop check while j<=k
            if i mod j = 0 then exit is_it_prime end if
            j := j + 2
          end loop check
          primeno := i
          suspend
        end begin is_it_prime
        i := i + 2
      end loop odd_numbers
end function primeno
```

B-10. PREDEFINED PROCEDURES AND FUNCTIONS

The procedures and functions introduced in this chapter and in B-11, are predefined in every implementation of PISA. Any implementation may feature additional predefined procedures and functions; their use in a program, however, results in a portability diagnostic message (cf. P-2.6.). All predefined procedures and functions are assumed to be declared in a virtual block surrounding the program. Hence, no conflict arises from a declaration redefining the same identifier within the program.

The formal parameter lists of some predefined procedures and functions introduce a few extensions to the parameter mechanism defined for user-declared PISA procedures and functions:

- If the formal parameter is a type, the actual parameter must be a type identifier.
- If the formal parameter accepts any type, the type of the actual parameter defines the type of the formal parameter.
- A value parameter may be restricted to a constant of a given type.
- The actual parameter list may consist of a variable number of parameters.

B-10.1. Ordinal functions

Ordinal functions may be applied to all discrete types except integer (yet to integer subrange types!). Their definition is based on the ordered set of values defined for every discrete type. "t" denotes a type identifier of a discrete type other than integer.

- | | |
|-----------|---|
| first(t) | returns the first (lowest) value of type t. |
| last(t) | returns the last (highest) value of type t. |
| succ(t,v) | returns the value succeeding the value v of type t.
The program is illegal if v=last(t). |

`pred(t,v)` returns the value preceding the value `v` of type `t`. The program is illegal if `v=first(t)`.

`ord(t,v)` returns an integer value which is the ordinal number of the value `v` in the enumeration of `t`'s root type. `v` must be a value of type `t`. `ord(t,first(t))=0` if `t` is an enumerated type; if `t` is a subrange of root type `r`, then `ord(t,v)=ord(r,v)`.

`val(t,n)` returns a value of type `t` which has the ordinal number `n` in the enumeration of `t`'s root type. Thus `val(t,ord(t,v))=v`. The expression denoted by `n` must be compatible with type integer. If `n` is not in the subrange `ord(t,first(t))..ord(t,last(t))`, the program is illegal.

Examples: assuming the type declarations

```

type day = (mon,tue,wed,thu,fri,sat,sun)
type workday = mon..fri
type holiday = sat..sun

first(day)      is mon
last(workday)   is fri
succ(day,mon)   is tue
pred(workday,thu) is wed
ord(holiday,sun) is 6      (!!)
val(holiday,5) is sat      (!!)
val(holiday,3) is illegal

```

E-10.2. String handling functions

The functions described in this chapter may be used for manipulating strings. "`s`", "`s1`", and "`s2`" denote expressions compatible with type string.

`len(s)` returns an integer value which is the actual length of `s`.

`index(s1,s2)` returns an integer value which is the starting position of the first occurrence of `s2` within `s1`. If `s2` is no substring of `s1`, 0 is returned. If `s2` is a nullstring, 1 is returned independent of the value of `s1`.

`xlate(s1,s2)` returns a string `s` which is the translation of `s1` according to the translation string `s2`. The translation is defined such that `len(s) = len(s1)` and `s<:i,1:> = s2<:ord(char,s1<:i,1:>)+1,1:>` for $1 \leq i \leq \text{len}(s1)$. Before translation, the string `s2` is padded with blanks to a length of `ord(char,last(char))+1`.

`strip(s)` returns a string which is the string `s` without leading and trailing blanks and tabulation characters (e.g. ASCII 9).

`cap(s)` returns a string which is the string `s` with all small letters converted to their capital equivalent.

`small(s)` returns a string which is the string `s` with all capital letters converted to their small letter equivalent.

Substrings are denoted by the substring construct introduced in E-3. Notice that whenever a predefined string handling function returns a string, this string is never longer than the string that is the first (or only) actual parameter to the function. The implications of this property are explained in E-2.4.

Examples:	<code>len('abc')</code>	is 3
	<code>len('')</code>	is 0
	<code>index('abcde','cd')</code>	is 3
	<code>index('abcde','dc')</code>	is 0
	<code>strip(' abcd ')</code>	is 'abcd'
	<code>cap('aBcdE')</code>	is 'ABCDE'
	<code>small('aBcdE')</code>	is 'abcde'

E-10.3. Numeric functions

Numeric functions are also referred to as arithmetic or mathematical functions and are applied to integer and real types. "n" denotes an expression of root type integer, "x" denotes an expression of root type integer or real. If the result of a function is defined to be of type real, it is always a value of type internal float as described in B-4.2.2.

<code>abs(x)</code>	returns the absolute value of <code>x</code> . The result value has the same type as <code>x</code> .
<code>trunc(x)</code>	returns an integer value which is the integer part of the value <code>x</code> . Truncation is always toward zero, so that <code>-trunc(x)=trunc(-x)</code> .
<code>round(x)</code>	returns an integer value which is the nearest integer to the value <code>x</code> , so that: if <code>x>=0</code> then <code>round:=trunc(x+0.5)</code> else <code>round:=trunc(x-0.5)</code> end if
<code>sqr(x)</code>	returns a real value which is the square root of the value <code>x</code> (<code>x≥0</code>).
<code>pi</code>	returns the real value which is the constant <code>pi</code> .
<code>sin(x)</code> <code>cos(x)</code> <code>tan(x)</code>	returns a real value which is the result of the trigonometric function for <code>x</code> expressed in radians.
<code>arctan(x)</code>	returns a real value <code>y</code> which is the arctangent of the value <code>x</code> , expressed in radians. $-(\pi/2) < \arctan(x) < +(\pi/2)$
<code>ln(x)</code>	returns a real value which is the natural logarithm (i.e. base <code>e</code>) of the value <code>x</code> . <code>x</code> must be greater than 0.
<code>log10(x)</code>	returns a real value which is the common logarithm (i.e. base 10) of the value <code>x</code> . <code>x</code> must be greater than 0.
<code>log2(x)</code>	returns a real value which is the binary logarithm (i.e. base 2) of the value <code>x</code> . <code>x</code> must be greater than 0.

E-10.4. Conversion and editing functions

Conversion functions serve to convert a value of a simple type to a string or vice versa. Editing functions are used to edit a numeric (integer or real) value according to a specified editing picture.

str(v)

The function "str(v)" returns a string which is the string representation of the value denoted by the expression v. v must be of a simple type. The string representation for a value v of root type t is defined as follows:

- t=integer: The significant digits representing the value v, preceded by a minus sign if v is negative. The value zero is represented as '0'.
- t=boolean: 'false' or 'true', resp.
- t=char: The character denoted by v (without surrounding quotes).
- t=other enumerated type: The value identifier which represents the value v.
- t=fixed: The significant digits representing the integer part of the value v, followed by a decimal point, followed by the fractional digits. If v is negative, the number is preceded by a minus sign. The number of fractional digits corresponds to the type definition of t. If the integer part is zero, '0' precedes the decimal point.
- t=float: A minus sign if the mantissa is negative, followed by 1 to m significant digits representing the mantissa, followed by the letter 'E', followed by the sign of the exponent, followed by two digits representing the exponent. The maximum number m of mantissa digits corresponds to t's type definition. If the mantissa contains more than one significant digit, a decimal point is inserted between the first and the second digit; the exponent is adjusted accordingly. If t is internal float (R-4.2.2.), m is equal to the highest m accepted in a float type definition (B-2.2.2.) on the actual implementation.
- t=string: The string denoted by v (without surrounding quotes).

Examples: assuming the declarations and assignments

```

type alpha = 'A'..'Z'
type day   = (Mon,Tue,Wed,Thu,Fri,Sat,Sun)
var x: fixed(-10000..+10000,2)
var y: float(-1E10..+1E10,6,1E-5)
x := -513.5
y := 893.15E5

```

str(13-7)	is '6'
str('A' < 'B')	is 'false'
str(first(alpha))	is 'A'
str(succ(day, Mon))	is 'Tue'
str(x)	is '-513.50'
str(y)	is '8.9315E+07'

conv(t,s)

Any value of a simple type *t* can be textually represented by a constant defined for type *t* (cf. B-2.2.). The constant is an identifier denoting a value of an enumerated type, an optionally signed number representing an integer or real value, or a literal string standing for a character or string value.

The function "conv" returns the value of type *t* that is textually represented by the string *s*. *t* must be a type identifier designating a simple type other than integer (yet an integer subrange type). *s* is an expression compatible with type string whose actual value is the textual representation of a value compatible with type *t*. If *s* represents a value not compatible with type *t*, the program is illegal. According to matching rules for identifiers (B-1.2.), an identifier denoting a value of an enumerated type may contain small or capital letters. A literal string represented by *s* must not be enclosed in quotes since, unlike literal strings that are part of the PISA program text, the literal string's boundaries are implicitly given by *s*. The string *s* must not contain any separators (B-1.4.).

Examples: assuming the declarations

```

type colour = (red,blue,Green,YELLOW)
type index  = 0..100
type weight = fix (0..1000,2)
type name   = string(5)

conv(colour,'red')      is red
conv(colour,'green')    is Green
conv(colour,'YELLOW')   is YELLOW
conv(colour,'bl'+ue')   is blue
conv(colour,'brown')    is invalid!
conv(index,'+72')       is 72
conv(weight,'03.250E+1') is 32.50
conv(char,'a')           is 'a'
conv(name,'abcdefg')     is 'abcde'
conv(name,'ab')          is 'ab'
```


tconv(t,s)

The function "tconv" (test conversion) returns the boolean value false when the function "conv" invoked with the same parameters would result in an illegal program, true otherwise. Thus, the program segment

```
if tconv(t,s) then x:=conv(t,s) end if
```

never results in an illegal program if t is the type of x and denotes a simple type other than integer.

edit(v,e)

The function "edit(v,e)" returns a string s which is the value of the expression v edited according to the string e which is called the editing picture. The root type of v must be integer or real. The length of s is always equal to the length of e which consists of a number of picture characters. e must be an entire constant of type string.

A picture character '.' specifies the location of the decimal point, a '9' the location of a decimal digit in the edited value v. Leading and trailing zeroes are inserted if v has less significant digits than specified in the editing picture; if v has more significant digits than specified in the editing picture, leading and trailing digits are truncated. The value is edited without a sign, even if it is negative. If the editing picture does not contain a '.', the decimal point is assumed to be immediately to the right of the rightmost picture character. The editing picture may not contain more than one '.'.

edit(37.38,'99.99')	is '37.38'
edit(52,'9999.99')	is '0052.00'
edit(2864.982,'99.9')	is '64.9'
edit(-58.1,'999.99')	is '058.10'
edit(85,'999')	is '085'
edit(85.62,'999')	is '085'

Picture characters 'Z' allow to replace leading zeroes by blanks, '*' to replace them by asterisks. An editing picture may only contain either 'Z' or '*', but not both. 'Z' and '*' are not allowed to the right of a '9' or '.'.

edit(15.4,'ZZZ9.99')	is ' 15.40'
edit(0.5,'ZZ9.9')	is ' 0.5'

```

edit(0,'ZZZ.9')          is '   .0'
edit(-37,'*****')       is '***37'
edit(21.80,'**999.99')   is '**021.80'

```

The picture characters '+' and '-' are used to specify the position of a sign. An editing picture may contain only one occurrence of '+' or '-'. A '+' indicates that a sign is always required (either + or -); a '-' indicates that a - sign is required for negative values, a blank replacing the sign if the value is zero or positive. The sign picture character may either precede all occurrences of '9', '.', 'Z', and '*', or succeed them. If it precedes them, the sign appears immediately preceding the first nonblank character in the edited string s (therefore called a floating sign); if it succeeds all '9', '.', 'Z', and '*', the sign appears exactly in the position of the sign picture character.

```

edit(-832.50,'-ZZZZ9.99') is ' -832.50'
edit(0.51,'+ZZ.99')       is ' +.51'
edit(14.6,'-999.9')       is ' 14.6'
edit(86.93,'+99.99')      is '+86.93'
edit(-23.45,'**9.99-')    is '*23.45-'
edit(987.65,'ZZZ.99+')    is '987.65+'
edit(5,'ZZ9+')            is ' 5+'

```

The picture characters ',', ':', '/', and ' ' (blank) are called insertion characters and serve to visually decompose an edited number. Any number of insertion characters can be inserted at any place within the editing picture. An insertion character appears in the edited string only if it is preceded by a digit 0 to 9 or the decimal point, otherwise it is replaced by the character preceding it in the edited string.

```

edit(100377,'99/99/99')   is '10/03/77'
edit(-581935,'ZZZ ZZZ ZZZ') is ' -581 935'
edit(4.3871,'ZZ:ZZ.ZZ:ZZ+') is ' 4.38:71+'
edit(1,'**,**.99')        is '****1.00'

```

If a value is to be edited in mantissa-exponent format, this can be specified by the exponent picture characters which must follow all picture characters for the mantissa. The mantissa can be edited as described above. The exponent picture characters are four contiguous 'E' denoting the space required for editing the exponent. An exponent is always edited as an 'E' followed by the sign of the exponent followed by a two digit number. The value of the exponent is adjusted so that the first significant digit of the mantissa appears in the position specified by the first '9', 'Z', or '*' in the editing picture.

```
edit+ (.12345E6, '.99999EEEE') is '.12345E+06'
edit (.12345E6, '99.999EEEE') is '12.345E+04'
edit (-12.345E-7, '-Z, Z9.99 EEEF') is '-1,23.45 E-08'
```

The rules for the composition of edit strings are summarized in the following productions which are not part of PISA's grammar:

```
edit_string    ::= left_part ["EEEE"]
left_part      ::= sign number_part |
                  number_part [sign]
number_part    ::= ({ "Z" } | { "*" } ) { "9" }
                  [ "." { "9" } ]
sign           ::= "+" | "-"
ins_char       ::= " , " | " : " | " / " | " "
```

where insertion characters (ins_char) are allowed anywhere within edit_string and where the number part must contain at least one picture character "Z", "*", or "9".

editz(v,e)

The function "editz" returns the same value as the function "edit" as long as the relation $v < 0$ is true (cf. B-4.2.5.). If this relation is false, a string containing len(e) blanks is returned.

B-10.5. Dynamic storage procedures

Dynamic storage procedures serve to allocate and free dynamic variables (B-2.4.).

new(p) allocates a new variable *v* and assigns its address to the pointer variable *p*. The variable *v* is of *p*'s associated type. If *p*'s base type is a record type with variants, enough space is allocated to hold any variant. If the variable cannot be allocated because insufficient storage space is available, "nil" is assigned to *p*.

free(p) remits the space occupied by the dynamic variable *p*→ to the pool of free space and assigns the address constant "nil" to the pointer variable *p*. A freed variable should no longer be referenced by any pointer still containing its address or by a variable bound to it.

E-10.6. Program control procedures

Program control procedures serve to stall, stop, and trace the execution of a program.

sleep(m) The execution of the program is stalled for *m* seconds. *m* is an expression compatible with type 0..60 (subrange of integer). This procedure is not intended to stall the execution of the program for precisely *m* seconds. The accuracy of the stalling time depends on the implementation.

stop(s) The execution of the program is discontinued. If the program is executing in "run mode", the error R001 is printed on the terminal connected to the program before the session is switched to "halt mode" (cf. C-5.). If the program is executing in "execute mode", the error X001 is produced and the session switched to "system mode" (cf. C-6.).

trace(m) The procedure "trace" serves to control the trace facility. The trace facility is only supported in "run mode" (cf. C-5.) and is initially switched off. Any invocation of the procedure trace from "execute mode" is ignored. *m* is an expression compatible with type 0..3 (subrange of integer); it indicates the trace mode to be set:

0 = no trace: The trace facility is switched off.

1 = fast trace: Before execution of each statement in the program, the line number of the text line where this statement begins is printed on the terminal. The line numbers are printed as defined by the following algorithm:

```

loop line_by_line
  loop number_by_number for i:=1 to 9
    printnr(line_#,tab(i*7+1))
    suspend
  end loop number_by_number
  print('')
end loop line_by_line

```

where "line_#" is the line number to print, and where "suspend" indicates suspension of the printing routine until the next line number is to be output.

2. = slow trace: The trace mode 2 is defined exactly as the trace mode 1 except that after a line number has been printed and before execution of this statement starts, the statement "sleep(1)" is executed. A user interrupt (C-1.2.4.) during the sleeping time terminates the sleep, sets the interruption location (C-5.4.3.) preceding the next statement, and switches the session to "halt mode".

3. = single statement trace: Before execution of each statement in the program, the statements

```
    printnr(line_#)
    inputln(i)
```

are executed. "line_#" is the line number where the next program statement begins. The input given (presumably a null string) is ignored; the "inputln" procedure serves only to implement a "step-by-step" trace. A user interrupt (C-1.2.4.) during the time the "inputln" waits for an input sets the interruption location preceding the next program statement to be executed and switches the session to "halt mode".

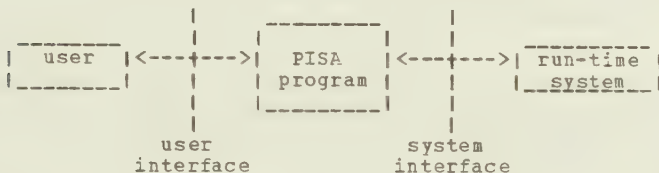
The statements entered for immediate execution (C-7.3.) are never traced.

E-10.7. Miscellaneous procedures and functions

areasize(a) This function returns an integer value which is the size (in bytes) of the area a. a must be an area parameter (B-4.3.).

E-11. INTERFACES OF A PISA PROGRAM

A PISA program has only two interfaces to communicate with its environment: one to its user, the other to the supporting run-time system. The interfaces are called user interface and system interface.



E-11.1. The user interface

The user of a PISA program is either a person operating a terminal connected to PISA or a program simulating terminal input and output. The latter is called pseudo terminal operation. It is of no concern to the PISA program whether its user is a person or another program. To simplify terminology, let us assume here that the user interface connects a real terminal to a PISA program.

The procedures and functions available for handling the user interface by a PISA program can be classified into:

1. Input procedures accept data entered at the terminal and pass it to the PISA program.
2. Output procedures transfer data from the PISA program to the terminal.
3. Control procedures and functions are used to control certain operations of the user interface and to get additional information from it.

The following user interface procedures and functions are predefined for every PISA implementation.

Input_procedures

`input(v)` The procedure "input" stalls the execution of the program and waits for input from the terminal. The input is a character string `s` containing all characters typed until but not including an end-of-line. "carriage return" (ASCII 13), "line feed" (ASCII 10), and "escape" (ASCII 27) are end-of-line characters and terminate an input. `v` must be an actual variable parameter of any simple type `t`. The value denoted by `s` is assigned to `v` by

`v := conv(t,s)`

If the conversion fails, the program is illegal. The definition of the function "conv" (B-10.4.) and of the type adaptation rules (B-2.5.1.) imply that the conversion never fails when the root type of `v` is "string".

`inputln(v)` The procedure "inputln" (input line) is defined exactly as the procedure "input", except that `v` must be of a string type and that the end-of-line character is the last character of the string assigned to `v`.

`cinputln(v)` The procedure "cinputln" (conditional input line) is defined exactly as the procedure "inputln", except that the program is never stalled. If there is no complete input line available from the terminal, a nullstring is assigned to `v`, otherwise the definitions for "inputln" apply.

Output_procedures

`print(...)` The procedure "print" may contain an actual parameter list with 1 to `n` value parameters. Starting with the first parameter in the list, the value of each parameter `p` is converted to its string representation by "str(p)" and then printed out on the terminal. After the last parameter has been printed, the necessary control characters are passed to the terminal to move the cursor to column 1 of the succeeding line (usually a CR/LF sequence).

`printnr(...)` The procedure "printnr" (print with no return) is defined exactly as the procedure "print", except that the cursor of the terminal remains behind the last character printed.

Control procedures and functions

`col` The function "col" returns an integer value which is the column of the next character (i.e. the present position of the cursor) on the actual output line. The leftmost column of a line is column 1.

`tab(m)` The function "tab" moves the cursor of the terminal to column `m` and returns a nullstring. `m` must be an expression compatible with type integer. If `m` ≤ `col`, the cursor is not moved. The common usage of the function "tab" is as a parameter to the procedures "print" and "printnr".

`echo(c)` The procedure "echo" serves to switch on and off the echoing of input data. If the value parameter `c` of type boolean is true, input data received from the terminal is echoed back to the terminal, if `c` is false, no data is echoed. Whenever a program terminates, whether normally or abnormally, the echo is automatically switched on. Initially, the echo is switched on.

`interrupt(c)` The procedure "interrupt" serves to enable and disable the user interrupt facility (cf. C-1.2.4.). If the value parameter `c` of type boolean is true, the user may signal a user interrupt, if it is false, all user interrupts are ignored. Whenever a program terminates, whether normally or abnormally, the user interrupt facility is automatically enabled. Initially, the user interrupt facility is enabled.

The user interface procedures and functions as defined above are based on an implementation of the user interface as described in E-3. In addition to the tasks mentioned so far, the user interface is also responsible to make provision for handling certain peculiarities of the terminal's hardware like code translation, delay time generation, etc. Since these tasks are not controlled by the PISA program but by the run-time system,

according information is passed to the user interface by means of a specific "sys" procedure call (see B-11.2. and Appendix B-II).

```
Examples:  input(a)
           input(colour)
           inputln(x)
           print('Your balance is ',b,' Dollars')
           print('')      (* causes a line skip only *)
           printnr('A=',a,tab(30),'B=',b)
           echo(answer='yes')
           interrupt(false)
```

B-11.2. The system interface

Besides the communication path to its user, a PISA program has another window to its environment: the system interface. The system interface is a message handler processing the data transfer between the PISA program and the run-time system. This interface is used by the PISA program to issue so called system service requests (abbreviated as SSR) to the run-time system. The one and only way to access the system interface from a PISA program is by executing a procedure statement invoking the predefined "sys" procedure.

A PISA program requests a system service whenever it is not able to perform this service by its own means. Among other usages, an SSR is issued for:

- Executing all I/O except to and from the user interface (which is not considered to be part of the run-time system).
- Getting the current date and time.
- Getting accounting information.
- Operating a pseudo terminal.
- etc.

A procedure statement designating the "sys" procedure has 1 to n parameters in its actual parameter list:

```
sys(p1)
sys(p1,p2)
:
:
sys(p1,p2,...,pn)
```

The first parameter p1 is the SSR code. It specifies the kind of service requested from the run-time system. p1 must be an entire constant of type integer.

Parameters p2 to pn serve for data exchange between the PISA program and the run-time system; p2 to pn are variable parameters unless the description of the SSR defines something different. Their types depend on the SSR code specified by p1 and will be defined in Appendix B-II. The compiler is not obliged to check the types of p2 to pn.

The SSRs are classified into three usage categories (nonrestricted, restricted, reserved) and into three portability categories (portable, semiportable, nonportable). The SSR code is a four digit integer constant; its first digit specifies the usage and portability category according to the following table.

	portable	semi- portable	non- portable
nonrestricted	1...	4...	7...
restricted	2...	5...	8...
reserved	3...	6...	9...

The usage category specifies the class of users that are allowed to compile a specific SSR. User privilege classes will be discussed in C-1.3.

- nonrestricted SSRs can be compiled by all users.
- restricted SSRs can be compiled by users that have restricted privileges or full privileges.

- reserved SSRs can only be compiled by users with full privileges.

Notice that the usage categories restrict the compilation of certain SSRs to certain classes of users but do not limit the right to execute such SSRs. An SSR that is contained in an external routine is issued under control of this routine which is available to all users with access rights to it (cf. C-4.7.).

The portability categories indicate the degree of portability for an SSR:

- portable SSRs are provided on every PISA implementation. The types of all formal parameters are given and their values have precise meanings.
- semiportable SSRs are provided on every PISA implementation. The types of all formal parameters are given but the value of at least one of them is only weakly defined. "Weakly defined" means that the purpose of a parameter is defined but an exact definition of its values is not given. If a parameter, for example, is of type string and specifies a file name, it is weakly defined because the syntax of file names is implementation dependent.
- nonportable SSRs are not available on every PISA implementation. The implementation handbook specifies their SSR codes together with a definition of the parameters.

Most of the data transferred between the PISA program and the PISA run-time system has a well defined type. The run-time system relies on that type and processes the data based on it. There is, however, data whose type is of no concern to the run-time system. A record written to or read from a peripheral device, for example, has no type as far as the run-time system looks at it; it is just a sequence of bytes transferred through the system interface. A formal area parameter (cf. F-7.) accepts an actual parameter of any type and is used for passing "typeless" areas to the run-time system. The implementation passes the address and the length of the area only.

Section D provides some guidelines for using "sys" procedure statements in different programming environments. To avoid misunderstandings, it is mentioned already here that the common user does usually not code "sys" procedure statements. He calls

external procedures with mnemonic names (such as date, open, read) to perform the desired tasks. These procedures were written by experienced programmers, check the actual parameters received, issue SSRs, and pass back the appropriate data to the invoking procedure.

The design of the system interface and of the system service requests introduced in Appendix B-II has been influenced by various operating systems. It is virtually impossible to associate all PISA interface characteristics with the source they originally stem from. However, it would be unfair not mentioning the operating system that had the greatest overall effect on PISA's interface structure and SSF concept: Digital Equipment's RSTS-E [DEC1] [DEC2]. Readers familiar with Bell's UNIX [Ritchie/Thompson,74] note a certain similarity with this operating system as well. This is presumably because UNIX' designers were also influenced by DEC's family of operating systems for the PDP-11 computer series.

Appendix B-I: Collected syntax of PISA

Lexical structure

```

letter      ::= "A"|"B"|"C"|"D"|"E"|"F"|"G"|"H"|"I"|"J"|"K" |
               "L"|"M"|"N"|"O"|"P"|"Q"|"R"|"S"|"T"|"U"|"V" |
               "W"|"X"|"Y"|"Z" |
               "a"|"b"|"c"|"d"|"e"|"f"|"g"|"h"|"i"|"j"|"k" |
               "l"|"m"|"n"|"o"|"p"|"q"|"r"|"s"|"t"|"u"|"v" |
               "w"|"x"|"y"|"z" |
               "#"|" $"|"%"

digit       ::= "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"

spec_symbol ::= "+" | "-" | "*" | "/" | "**" | "=" |
               "<" | "<" | ">" | "<=" | ">=" | ">" |
               "(" | ")" | "." | "(" | "*" | "*" | ":" |
               "." | "," | ":" | ";" | ":" | ":" |
               "<:" | ">:" | "<:" | ">:" | "<:" | ">:" |

word_symbol ::= "action"|"and"|"area"|"array"|"begin"|"bind" |
               "case"|"cond"|"const"|"coroutine"|"datab"|"div" |
               "downto"|"else"|"end"|"exit"|"external"|"fix" |
               "float"|"for"|"forward"|"function"|"goto" |
               "if"|"in"|"loop"|"main"|"mod"|"nil"|"not" |
               "of"|"or"|"otherwise"|"packed"|"procedure" |
               "record"|"rep"|"set"|"string"|"suspend" |
               "then"|"to"|"type"|"var"|"vary"|"when"|"while"

identifier  ::= letter {letter | digit | "_"}

..._id      ::= identifier

integer     ::= digit {digit}

number      ::= integer "." integer |
               integer ["." integer]
               ("E"|"e") ["+"|"-"] integer

lit_string  ::= '"' {character} '"'

cond_entry  ::= "-" {"Y"|"y"|"N"|"n"|"-" } ""

action_entry ::= "" {"X"|"x"|"-" } {"X"|"x"|"-" } ""

```

Data types

```

type_dcl    ::= "type" type_id "=" type_def

type_def    ::= simple_type | struc_type | pointer_type |
               type_id

simple_type  ::= discrete_type | real_type | string_type

discrete_type ::= enum_type | subrange_type

enum_type   ::= "(" constant_id {"," constant_id} ")"

subrange_type ::= entire_const ".." entire_const

```

```

real_type      ::= "fix" "(" range "," entire_const ")" |
                  "float" "(" range "," entire_const ","
                      entire_const ")"
range          ::= entire_const ".." entire_const
string_type    ::= "string" "(" entire_const ")" ["vary"]
struc_type     ::= ["packed"] (array_type | record_type | set_type)
array_type     ::= "array" "(" index_type {"," index_type} ")"
                  "of" compnt_type
index_type     ::= discrete_type | type_id
compnt_type    ::= type_def
record_type    ::= "record"
                  [fixed_part]
                  [variant_part]
                  "end" "record"

fixed_part     ::= {field_dcl}
variant_part   ::= "case" tag
                  {when_vnt}
                  [otherwise_vnt]
                  "end" "case"

when_vnt       ::= "when" when_list ":" {field_dcl}
otherwise_vnt  ::= "otherwise" ":" {field_dcl}
when_list      ::= when_item {"," when_item}
when_item      ::= entire_const [".."] entire_const
field_dcl      ::= field_id {"," field_id} ":" type_def
tag            ::= field_id | type_id
set_type       ::= "set" "of" base_type
base_type      ::= discrete_type | type_id
pointer_type   ::= ">" assoc_type
assoc_type     ::= ">" type_id

```

Constants and variables

```

constant_dcl  ::= "const" constant_id "="
                  (entire_const | type_def const_list)
const_list    ::= "(:" [const_item {"," const_item}] ":)"
const_item    ::= entire_const [".."] entire_const | const_list
constant      ::= entire_const | substr_const | comp_const
entire_const  ::= literal_const | ["+"|"-"] constant_id
literal_const ::= integer_const | real_const | lit_string | "nil"
integer_const ::= ["+"|"-"] integer
real_const    ::= ["+"|"-"] number
substr_const  ::= constant "<:" expression [{"," expression}] ">"
comp_const    ::= ["+"|"-"] (indexed_const | field_const)
indexed_const ::= constant "(" expression {"," expression} ")"
field_const   ::= constant "." field_id
variable_dcl  ::= "var" variable_id {"," variable_id} ":"
                  type_def |
                  "bind" variable_id "to" variable

```



```

variable      ::= entire_var | substr_var | comp_var
                referencd_var
entire_var    ::= variable_id
substr_var    ::= variable "<:" expression ["," expression] ">:"
comp_var      ::= indexed_var | field_var
indexed_var   ::= variable "(" expression {"," expression} ")"
field_var     ::= variable "." field_id
referencd_var ::= variable "->"

```

Expressions

```

simple_factor  ::= variable | constant | function_des |
                "(" expression ")" | value_constr |
                sign_op simple_factor
factor        ::= simple_factor | factor "***" simple_factor
term          ::= factor | term mult_op factor
sum           ::= term | sum add_op term
relation      ::= sum | sum rel_op sum
negation      ::= relation | "not" relation
conjunction   ::= negation | conjunction "and" negation
expression    ::= conjunction | expression "or" conjunction
value_constr  ::= type_id comp_list
comp_list     ::= "(" [comp_val {"," comp_val}] ")"
comp_val      ::= expression [ "." expression ] | comp_list
sign_op       ::= "+" | "-"
mult_op       ::= "*" | "/" | "div" | "mod"
add_op        ::= "+" | "-"
rel_op        ::= ["rep"] comp_op | "in"
comp_op       ::= "=" | "<>" | "<" | ">" | "<=" | ">="
function_des  ::= function_id [act_parmlist]
act_parmlist  ::= "(" actual_parm {"," actual_parm} ")"
actual_parm   ::= expression | variable

```

Blocks

```

block         ::= decl_part stmt_part
decl_part     ::= {declaration}
declaration   ::= variable_dcl | constant_dcl | type_dcl |
                procedure_dcl | function_dcl
stmt_part     ::= {statement}

```

Statements

```

statement     ::= {label_dcl} {simple_stmt | struc_stmt}
label_dcl     ::= "*" label_id {"," label_id} ":"

```

```

simple_stmt      ::= assign_stmt | proc_stmt | goto_stmt |
                    exit_stmt | suspend_stmt
assign_stmt     ::= (variable | function_id) "!=" expression
proc_stmt      ::= procedure_id [act_parmlist]
goto_stmt      ::= "goto" label_id
exit_stmt      ::= "exit" block_id
suspend_stmt    ::= "suspend"
struc_stmt     ::= compound_stmt | conditnl_stmt | repetit_stmt
compound_stmt   ::= "begin" block_id
                    begin_block
                    "end" "begin" block_id

begin_block    ::= block
conditnl_stmt  ::= if_stmt | case_stmt | detab_stmt
if_stmt        ::= "if" expression
                    "then" then_block
                    ["else" else_block]
                    "end" "if"

then_block     ::= block
else_block     ::= block
case_stmt      ::= "case" expression
                    {when_constr}
                    [otherw_constr]
                    "end" "case"

when_constr    ::= "when" when_list ":" when_block
otherw_constr  ::= "otherwise" ":" otherw_block
when_block     ::= block
otherw_block   ::= block
detab_stmt     ::= "detab"
                    cond_part
                    action_part
                    "end" "detab"

cond_part      ::= condition {condition}
condition      ::= "cond" expression cond_entry
action_part    ::= action {action}
action         ::= "action" action_entry
                    action_block

action_block   ::= block
repetit_stmt   ::= "loop" block_id [ for_constr | while_constr ]
                    loop_block
                    "end" "loop" block_id

for_constr     ::= "for" control_var "!=" initial_value
                    ("to"|"downto") final_value

while_constr   ::= "while" expression
control_var    ::= identifier
initial_value  ::= expression
final_value    ::= expression
loop_block     ::= block

```

Procedure_declarations

```

procedure_dcl ::= proc_def | external_proc | forward_proc
proc_def      ::= ["coroutine"] "procedure" procedure_id
                [form_parmlist]
                proc_block
                "end" "procedure" procedure_id
proc_block    ::= block
form_parmlist ::= "(" formal_parm {",", " formal_parm} ")"
formal_parm   ::= ["var"] identifier ":" type_def |
                "area" identifier
external_proc ::= "procedure" procedure_id "external"
                [parm_typelist]
parm_typelist ::= "(" parm_type {"", " parm_type} ")"
parm_type     ::= ["var"] type_def | "area"
forward_proc  ::= "procedure" procedure_id "forward"
                [parm_typelist]

```

Function_declarations

```

function_dcl  ::= function_def | external_func | forward_func
function_def  ::= ["coroutine"] "function" function_id
                [form_parmlist] ":" type_def
                func_block
                "end" "function" function_id
func_block    ::= block
external_func  ::= "function" function_id "external"
                [parm_typelist] ":" type_def
forward_func  ::= "function" function_id "forward"
                [parm_typelist] ":" type_def

```

Programs_and_external_routines

```

compilation  ::= program | ext_routine
program      ::= "main" proc_def
ext_routine  ::= "external" (proc_def | function_def)

```

Appendix B-II: System service requests

=====

As introduced in B-11.2., a PISA program issues a system service request (SSR) by calling the "sys" procedure which handles the system interface. This appendix defines the system services available to a PISA program together with the parameters of the "sys" procedure statement requesting it.

It should be noted that this appendix does probably not introduce all SSRs provided by a PISA implementation. It is just a list of the SSRs that are likely to be available. A first implementation of PISA is a necessary prerequisite for a final version of this appendix.

The SSRs are defined in the sequel ordered on ascending SSR codes. The SSR code is the first actual parameter in every "sys" procedure statement (cf. B-11.2.). The parameters 2 to n are variable parameters denoted by p2 to pn unless the description of the SSR defines something different. The type definition is given for every variable parameter; area parameters are indicated by the symbol "area" replacing the type definition.

Portable system service requests (nonrestricted)

1000-----get_implementation_parameters

```
p2 record
    name:                string(50) vary
    version:             0..100
    modification:        0..1000
end record
```

Returns the name, the actual version, and the modification level of the PISA implementation. The name of an implementation is unique among all PISA implementations and indicates the name of the computer series, the operating system, and the implementor (e.g. 'IBM370,MVS,XYZ-CORP.')

1001_____get_installation_parameters

```
p2 record
    name:                string(50) vary
    installation_#:       string(15) vary
end record
```

Returns the name of the installation (e.g. 'MAD-BIT COMPUTATION CENTRE, NOWHEREVILLE') and the installation number (e.g. '14164823148'). The installation number should be unique among all installations in the world. An easy solution for assigning unique installation numbers is to use the telephone number of the installation's operator as dialed for international calls (the installation number would, of course, not change if the telephone number changes).

The main purpose of the "get installation parameters" SSR is to prevent illegal usage of licensed software: any program can get information about the installation it is running on, compare it with the figures established when it was installed, and take measures if it detects an illegal usage.

1002_____get_date_and_time

```
p2 record
    year:                -1..2100
    month:               -1..12
    day:                 -1..31
    hour:                -1..23
    minute:              -1..59
    second:              -1..59
end record
```

Returns the actual date and time. If the date is not available on the implementation, "year", "month", and "day" all contain -1. The year is returned with millennium and century (e.g. 1978). If any of the time fields is not available, it contains -1.

1003_____get_program_size

```
p2 record
    act_size:            -1..10000
    max_size:            -1..10000
end record
```

Returns the actual memory space occupied by the program issuing the SSR as well as the maximum size to which it can expand. Both

figures are given in kilobytes (1 kilobyte = 1024 bytes). A value of -1 indicates that the figure is not available on the implementation.

1004-----get_communication_string

p2 (* comm_string *) string(255) vary

The SSR 1004 can be used to access the communication string which is assigned to parameter 2. Issuing an SSR 1004 does not change the value of the communication string.

If the actual program has been activated by the SSR 4000, the value of the communication string is the value set by the SSR 4000. If the program has been activated by the PISA run-time system, the value of the communication string is set by the run-time system as defined in Section C.

1005-----get_terminal_characteristics

```
p2 record
    term_type:           0..3
    line_width:          1..255
    page_size:           0..255
    hardware_tab:         boolean
    tab_step:            0..255
    mode_of_op:          0..1
end record
```

The characteristics of the terminal connected to a user interface can be inquired by the SSR 1005:

term_type: Indicates the type of the terminal:
 0 = pseudo terminal
 1 = cathode ray tube
 2 = storage tube
 3 = printing

line_width: The number of characters per line.

page_size: The number of lines the screen is able to hold. A page size of zero indicates an unlimited page size (pseudo terminals, printing terminals).

hardware_tab: "true" if the terminal provides a hardware tabulation mechanism (electronic or mechanical), "false" otherwise.

tab_step: Specifies the step-width of tabulations so that the n-th tab stop is at column $n \times \text{tab_step}$, where the leftmost column of an output line is column 1. A step-width of zero indicates that no tabulation is performed, i.e., that all tabulations are ignored.

mode_of_op: Indicates the data transmission mode between the user interface and the terminal:
 0 = half duplex
 1 = full duplex

1006 set terminal characteristics

```
p2 record
    subcode:           1..10
    s:                 string(256) vary
    i:                 0..10000
    completion_code:   0..2
end record
```

When a terminal is connected to the PISA system, certain terminal characteristics are assumed by default (cf. C-2.1.). The SSR 1006 can be used to change these default characteristics to the ones of the actual terminal connected. The definition of this SSR is based on an implementation of the user interface as described in E-3.

The subcode specifies the characteristic to be modified according to the information contained in the fields s and i. The run-time system sets the completion code to 0 if the characteristic has been successfully modified, to 1 if any of the parameter fields contains an illegal value, or to 2 if the characteristic specified by the subcode cannot be modified on this implementation. The subcode 4 may not be specified if the terminal connected is a pseudo terminal and would result in a completion code 1.

subcode action taken

- 1 The input translation table is replaced by s. The length of s must be 256.
- 2 The output translation table is replaced by s. The length of s must be 256.
- 3 The delay time for the character in $s<:1,1>$ is set to i milliseconds.

- 4 The terminal type is assumed according to i:
 i = 1: cathode ray tube terminal
 i = 2: storage tube terminal
 i = 3: printing terminal
- 5 The line width is assumed to be i characters.
- 6 The page size is assumed to be i lines. i=0 means unlimited page size (pseudo terminals or printing terminals).
- 7 The transfer rate is assumed to be i characters per second (not i bauds!).
- 8 If i=1, the terminal is expected to have a hardware tabulating mechanism; if i=0, no tabulating feature is assumed.
- 9 The field i specifies the step-width of tabulations so that the n-th tab stop is assumed at column n*i, where the leftmost column of an output line is column 1. If hardware tabulation has been specified for the terminal (see code 8 above), the tabulation step-width should correspond to the tabulations actually set. If no hardware tabulation is assumed, the step-width may be set as desired. A step-width of zero indicates that no tabulation is to be performed, i.e., that all tabulation characters have to be ignored.
- 10 If i=0, the mode of the terminal's operation is assumed to be half duplex (i.e. local echo); if i=1, the terminal is assumed to operate in full duplex (i.e. remote echo) mode.

Portable system service requests (restricted)

2000_____create_pseudo_terminal

p2 (* pseudo_term_# *) 0..1000

This SSR is used for creating a pseudo terminal. A pseudo terminal controls a PISA session in the same way as a real terminal connected to PISA (cf. Section C). In opposition to a real terminal operated by a person, a pseudo terminal is operated by the PISA program issuing the "create pseudo terminal" SSR.

The PISA program issuing this SSR is called the controlling program; the session controlled via the pseudo terminal is the controlled session. After successful completion of this SSR, the controlled session is in login state (cf. C-2.).

If the pseudo terminal has been successfully created and is ready for operation, its number is returned in parameter 2. This number is allocated at the run-time system's discretion and is in the subrange 0..1000 of integer. # number 0 specifies that the pseudo terminal could not be created. The implementation defines the maximum number of pseudo terminals that can be created by a single PISA program as well as the total number of pseudo terminals available.

2001 delete pseudo terminal

```
p2 (* pseudo_term_# *)      0..1000
```

A pseudo terminal previously created by SSR 2000 can be deleted by the SSR "delete pseudo terminal". The pseudo terminal number (parameter 2) must specify the number obtained from SSR 2000. Deleting a pseudo terminal is equivalent to disconnecting a real terminal. Just as a real terminal can be disconnected at any time, a pseudo terminal can be deleted at any given time after it has been created (cf. C-2.).

2002 enter data on pseudo terminal

```
p2 record
    pseudo_term_#:      0..1000
    accepted:           boolean
end record
```

```
p3 (* text *)           string(255) vary
```

A PISA program that has previously created a pseudo terminal by SSR 2000 can write on this pseudo terminal by issuing the SSR 2002. The pseudo terminal number (pseudo_term_#) must specify the number obtained from SSR 2000. Parameter 3 designates the text to be written on the pseudo terminal. The text includes all control characters that would be typed on the keyboard of a real terminal for entering the same text (e.g. tabulations, carriage returns, line feeds, etc.).

If the pseudo terminal waits for input when this SSR is issued, the text is entered on the pseudo terminal and the field "accepted" set to true; if the pseudo terminal does not wait for

input, the text in parameter 3 is ignored and the field "accepted" set to false.

If the text passed is a nullstring, no data is actually written on the pseudo terminal. Nevertheless, a nullstring is only "accepted" when the pseudo terminal waits for input exactly as described above. Therefore, the SSR 2002 can also be used to check if the pseudo terminal waits for input.

Section C defines the PISA session. Since it does not matter whether the user interface connects a real or a pseudo terminal to PISA, all definitions given there apply to both types of terminals. This implies that as long as the echo switch is set to "on" (cf. B-11.1.) and the pseudo terminal's mode of operation is "full duplex", the data entered on the pseudo terminal is echoed back to it.

2003-----read data from pseudo terminal

```
p2 (* pseudo_term_# *)      0..1000
p3 (* text *)                string(255) vary
```

A PISA program that has previously created a pseudo terminal by SSR 2000 can read data from this pseudo terminal by issuing the SSR 2003. The pseudo terminal number (pseudo_term_#) must specify the number obtained from SSR 2000. Let *s* be the sequence of characters written on the pseudo terminal by the controlled PISA session. Then we define *s* = *r w*, where *r* represents the part of *s* that has already been read by the SSR 2003 and *w* the part of *s* that waits to be read. If *w* reaches a certain implementation dependent length, the controlled PISA session is stalled by the PISA run-time system in order to prevent buffer overflow.

When this SSR is issued, *w* is assigned to parameter 3, *r*, as defined above, then becomes *r* concatenated with *w*, and *w* becomes that right part of *w* that has not been assigned to parameter 3 (only unequal to a nullstring if the length of *w* was greater than 255). Consequently, if *w* is empty when the "read data from pseudo terminal" SSR is issued, a nullstring is assigned to parameter 3.

The data read from the pseudo terminal includes all characters as if they were written on a real terminal. This includes carriage returns, line feeds, null characters, etc.

2004-----synchronize_with_pseudo_terminal

```
p2 record
    pseudo_term_#:      0..1000
    wait_time:          1..60
    waiting_for_input:   boolean
    output_to_read:      boolean
end record
```

A PISA program that has previously created a pseudo terminal by an SSR 2000 may synchronize with the controlled session by issuing the SSR 2004. The pseudo terminal number (pseudo_term_#) must specify the number obtained from SSR 2000. After the "synchronize with pseudo terminal" SSR has been issued by a program, this program is stalled until the wait time is exhausted, or either the pseudo terminal waits for input or has produced an output that has not yet been read by SSR 2003. The wait time is specified in seconds. When control returns from SSR 2004, the boolean field "waiting_for_input" indicates whether the pseudo terminal waits for input or not, the field "output_to_read" specifies whether there is data to be read from the pseudo terminal or not.

Portable system service requests (reserved)

3000-----logout

The SSR code is the only parameter of this SSR.

The "logout" SSR serves to terminate a session. Terminating a session causes the following actions:

- (1) Termination of the program issuing the SSR 3000.
- (2) Implementation dependent clean-up.
- (3) If the session is in the attached state, it is switched to the login state; if the session is in the detached state, it is switched to the idle state (cf. C-2.).

Control never returns to the program issuing a "logout" SSR.

3001-----detach

The SSR code is the only parameter of this SSR.

The "detach" SSP can be issued by a program to switch the session from the attached to the detached state (cf. C-2.). After the state has been switched, control returns to the program issuing the SSR. This implies that the PISA statement following the "detach" request already runs in the detached state.

The terminal originally connected to the attached session is disconnected by the "attach" request. The PISA run-time system then treats this terminal as if it would just have been connected to PISA (see C-2.). That is, it tries to switch an idle session to the login state and to connect the terminal to this session. As defined in C-2., the terminal can then be disconnected or used to login the new session.

The "detach" SSR never results in an error. If the session wherein it is issued is not in the attached state, this SSR is ignored.

3002-----attach

```
p2 record
      session_no:      1..1000
      completion_code: 1..3
end record
```

The "attach" SSR can be used to connect a terminal to a session running in the detached state. The session wherein this SSR is issued must be running in the attached state under the same account as the detached session specified by the field "session_no". If these conditions are satisfied, the following steps are performed by the PISA run-time system:

- (1) The program issuing the "detach" SSR is terminated.
- (2) The "logout" system program (cf. C-3.2.2.) is activated for the session wherein the "attach" SSR has been issued. Hence, this session is in the login state after this step.
- (3) The session is switched from the login state to the idle state.
- (4) The session specified by the field "session_no" is switched from the detached to the attached state.

- (5) The terminal originally connected to the session that issued the "attach" SSR is connected to the session switched to the attached state in step (4).

The steps described above show that control does not return to the program issuing a successful "attach" SSR. If control returns, no state switching and program termination has been performed. In this case, the field "completion_code" indicates the reason for the failure to attach the session:

- 1 = Session issuing the "attach" SSR is not in the attached state.
- 2 = Session specified by field "session_no" is not running under the same account as the session issuing the "attach" SSR or is nonexistent.
- 3 = Session specified by field "session_no" is not in the detached state.

Semiportable system service requests (nonrestricted)

4000_____chain to program

```
p2 record
    program:          string(16) vary
    comm_string:      string(255) vary
end record
```

Issuing this SSR leads to termination of the actual program and to transferring control to another program. The field "comm_string" (communication string) serves to pass a text to the program chained to. This program can access the communication string with the SSR 1004. If a "chain to program" SSR is issued in "run mode" activated from "edit mode" (cf. Section C), a run-time error results (C-5.4.1.4.). This implies that the SSR 4000 may only be used in programs selected from system mode.

After the actual program has been terminated by this SSR, the session is switched back to system mode. Then, the designated program is loaded and gains control as will be defined under "basic structure of the system mode" in C-3.1.

This SSR is only semiportable because the syntax of the field "program" is undefined.

4001_____get_accounting_data

```
p2 record
    account:          string(30) vary
    elapsed_time:     -1..100000000
    cpu_time:         -1..100000000
    cpu_load:         -1..100000000
    io_activity:      -1..100000000
end record
```

This SSR returns accounting data of the session wherein it is issued. A value of -1 means that the particular figure is not available on the implementation. The fields returned contain:

account: The account as entered at login or a nullstring if the session is in the login state (cf. C-2.2.). In the latter case, the values of all other fields are undefined.

elapsed_time: The elapsed time since login in 1/100 seconds.

cpu_time: The CPU time used since login in 1/100 seconds.

cpu_load: The program size integrated over the CPU time measured in kilobytes*seconds (1 kilobyte = 1024 bytes).

io_activity: The number of physical blocks transferred to or from peripheral devices other than the terminal connected to the user interface.

This SSR is only semiportable because the syntax of the field "account" is undefined.

4002_____get_session_parameters

```
p2 record
    session_no:      1..1000
    privileges:      0..2
    state:           0..2
    line:            string(10) vary
end record
```

This SSR returns parameters of the session wherein it is issued:

session_no: The number of the session.

privileges: 0 = the session has no privileges
 1 = the session has restricted privileges
 2 = the session has full privileges

 state: 0 = the session is in the login state
 1 = the session is in the attached state
 2 = the session is in the detached state

 line: The identification of the line connecting the real
 or pseudo terminal to the user interface of the
 session. If the session is in the detached state,
 the value of "line" is a nullstring.

This SSR is only semiportable because the syntax of the line identification is implementation dependent.

4003_____get active sessions

```

p2 record
    cntl:                1..3
    session_no:          1..1000
    state:               0..2
    mode:                0..4
    line:                string(10) vary
    program:             string(16) vary
end record
  
```

This SSR serves to get information about all sessions running under the same account as the session issuing this SSR. The information returned is:

session_no: The number of the session.

 state: 0 = the session is in the login state
 1 = the session is in the attached state
 2 = the session is in the detached state

 mode: 0 = the session is in system mode
 1 = the session is in edit mode
 2 = the session is in halt mode
 3 = the session is in run mode
 4 = the session is in execute mode

 line: The identification of the line connecting the real
 or pseudo terminal to the user interface of the
 session. If the session is in the detached state,
 the value of "line" is a nullstring.

program: The program running in the session or a nullstring
if the session is neither in the run mode nor in
the execute mode.

If the session issuing this SSR is denoted by *s*, and the sessions running under the same account as *s* are sessions *s1..sn*, then the *i*-th SSR 4003 returns information about session *si*. The ordering of the sessions is undefined. If the field "cntl" has the value 1 when this SSR is issued, *i* is set to 1 before the session information is retrieved. The value of the field "cntl" is set by the run-time system as follows:

- to 2 for all *i* ≤ *n*
- to 3 for all *i* > *n*

The session *s* is also one of the sessions *s1..sn*. Hence, this SSR always returns information of at least one session.

This SSR is only semiportable because the syntax of the fields "line" and "program" is undefined.

4010_____open_standard_sequential_file

```
p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record

p3 record
    filename:         string(50) vary
    proc_mode:        0..2
    shared_acc:        boolean
    record_length:     1..100000
    recs_per_block:    0..10000
    blk_per_cluster:   0..100000
    device_cntl:       string(20) vary
end record
```

The "open standard sequential file" SSR serves to initialize processing of a standard sequential file. It accepts the fields of parameter 3 and returns parameter 2. The meaning of the fields are:

file_#: When the file has been successfully opened
 (completion_code=0), the number of the file is

returned in this field. The file number is allocated at the run-time system's discretion.

completion_code: Is returned to the PISA program and specifies:

- 0 = file successfully opened
- 1 = input or update file not found
- 2 = output file already exists under file name specified
- 3 = protection violation
- 4 = file in use and no shared access
- 5 = illegal file name for output file
- 6 = open error, field "diagnostic" contains error message
- 7..10 not used by this SSR

diagnostic: If completion_code=6, this field contains a diagnostic message, otherwise its value is undefined.

filename: Indicates the name under which the file to process is known to the run-time system (external name).

proc_mode:

- 0 = file exists and is read only (input file)
- 1 = file is created (output file)
- 2 = file exists and is updated by appending records to its end (update file)

shared_acc: Specifies whether the program requests exclusive access to the file (shared_acc = false) or allows simultaneous shared access by several programs (shared_acc = true). Shared access can only be specified for input files.

record_length: Gives the number of bytes in one logical (internal) record.

recs_per_block: Defines the blocking factor of the file. If this information is not needed for opening the file, the value of "recs_per_block" must be 0.

blk_per_cluster: Indicates the number of blocks to be allocated per space allocation on secondary storage media as disks or drums. If this information is not required for opening the file or if the program does not want to specify the clustersize explicitly, the value of "blk_per_cluster" must be 0.

`device_cntl:` Contains device dependent control information as the type of printer or card punch control codes, density for tapes, the number of the file on a multifile tape reel, label type, etc. If there is no device dependent control information required for opening the file, the value of "device_cntl" must be a nullstring.

Several of the fields in this system service request depend in their format and possible values on the run-time system of the implementation. The format of the file name, for example, is given by the data management subsystem of the run-time system and, unfortunately, no attempt has been made yet to standardize file names. The values of other fields like record length, records per block, and device control are restricted by the run-time system and by the hardware of the devices. Therefore, the "open standard sequential file" SSR is only semiportable.

4011-----close_standard_sequential_file

```
p2 record
    file_#:          1..1000
    completion_code: 0..10
    diagnostic:       string(30) vary
end record
```

The "close standard sequential file" SSR is used to close a file previously opened by SSP 4010. The file number (file_#) must specify the number received by SSP 4010. If an open file is not closed before the program terminates, the program is illegal.

The completion code is returned to the PISA program and specifies:

- 0 = file successfully closed
- 1 = close error, field "diagnostic" contains error message
- 2..10 not used by this SSR

If completion_code=0, the value of the field "diagnostic" is undefined.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4012 read standard sequential file

```

p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record

p3 (* record area *)    area

```

This SSR serves to read the next record from a standard sequential file previously opened as input file by SSR 4010. The file number (file_#) must specify the number received from SSR 4010. The size of the area denoted by parameter 3 must be equal to the record length of the file.

The completion code is returned to the PISA program and specifies:

- 0 = record successfully read
- 1 = no more records in file (end of file)
- 2 = read error, field "diagnostic" contains error message
- 3..10 not used by this SSR

If completion_code <> 2, the value of the field "diagnostic" is undefined.

Once the end of a standard sequential file has been reached, every succeeding invocation of this SSR results in a completion code 1.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4013 write standard sequential file

```

p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record

p3 (* record area *)    area

```

This SSR serves to write the next record to a standard sequential file previously opened as output or update file by SSR 4010. The file number (file_#) must specify the number received from SSR

4010. The size of the area denoted by parameter 3 must be equal to the record length of the file.

The completion code is returned to the PISA program and specifies:

- 0 = record successfully written
- 1 = no more space on file to hold record
- 2 = write error, field "diagnostic" contains error message
- 3..10 not used by this SSR

If completion_code<>2, the value of the field "diagnostic" is undefined.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4020_____open_standard_direct_file

```
p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record

p3 record
    filename:         string(50) vary
    proc_mode:        0..2
    shared_acc:        boolean
    record_length:     1..100000
    recs_per_block:    0..10000
    blk_per_cluster:   0..100000
    device_cntl:       string(20) vary
end record
```

The "open standard direct file" SSR serves to initialize processing of a standard direct file. It accepts the fields of parameter 3 and returns parameter 2. The meaning of the fields are:

file_#: When the file has been successfully opened (completion_code=0), the number of the file is returned in this field. The file number is allocated at the run-time system's discretion.

completion_code: Is returned to the PISA program and specifies:

- 0 = file successfully opened
- 1 = input or update file not found
- 2 = output file already exists under file name specified
- 3 = protection violation
- 4 = file in use and no shared access
- 5 = illegal file name for output file
- 6 = open error, field "diagnostic" contains error message
- 7..10 not used by this SSR

diagnostic: If completion_code=6, this field contains a diagnostic message, otherwise its value is undefined.

filename: Indicates the name under which the file to process is known to the run-time system (external name).

proc_mode:

- 0 = file exists and is read only (input file)
- 1 = file is created (output file)
- 2 = file exists and is updated (update file)

shared_acc: Specifies whether the program requests exclusive access to the file (shared_acc = false) or allows simultaneous shared access by several programs (shared_acc = true). Shared access can only be specified for input and update files (see also note to SSR 4022).

record_length: Gives the number of bytes in one logical (internal) record.

recs_per_block: Defines the blocking factor of the file. If this information is not needed for opening the file, the value of "recs_per_block" must be 0.

blk_per_cluster: Indicates the number of blocks to be allocated per space allocation on the storage medium holding the file. The value of this field must be 0 when an input or update file is opened or if the program does not want to specify the clustersize explicitly.

device_cntl: Contains device dependent control information.
 If there is no device dependent control
 information required for opening the file, the
 value of "device_cntl" must be a nullstring.

This SSR is only semiportable because of the same reasons as
 indicated for SSR 4010.

4021-----close_standard_direct_file

```
p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record
```

The "close_standard_direct_file" SSR is used to close a file
 previously opened by SSR 4020. The file number (file_#) must be
 the number received by SSR 4020. If an open file is not closed
 before the program terminates, the program is illegal.

The completion code is returned to the PISA program and
 specifies:

- 0 = file successfully closed
- 1 = close error, field "diagnostic" contains error message
- 2..10 not used by this SSR

If completion_code=0, the value of the field "diagnostic" is
 undefined.

This SSR is only semiportable because the syntax of the error
 diagnostic is undefined.

4022-----read_standard_direct_file

```
p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record

p3 (* record number *)    1..1000000000

p4 (* record area *)      area
```

This SSR serves to read the record specified by parameter 3 from a standard direct file previously opened by SSR 4020. The file number (file_#) must specify the number received from SSR 4020. The size of the area denoted by parameter 4 must be equal to the record length of the file. The first record in a standard direct file has record number 1.

A record can be read from a direct file opened as input, output, or update file. The record specified must, however, exist on the file before it may be read in. After a record has been read from an update file, the block wherein this records resides is locked. A locked block cannot be accessed by another program. The block remains locked until the program rewrites the record (SSR 4023), unlocks it (SSR 4024), or closes the file (SSR 4021). If an implementation of PISA does not support block locking, it must reject shared access to direct update files.

The completion code is returned to the PISA program and specifies:

- 0 = record successfully read
- 1 = record does not exist
- 2 = block locked (see above)
- 3 = read error, field "diagnostic" contains error message
- 4..10 not used by this SSR

If completion_code <> 3, the value of the field "diagnostic" is undefined.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4023 ----- write_standard_direct_file

```
p2 record
    file_#:          1..1000
    completion_code: 0..10
    diagnostic:      string(30) vary
end record

p3 (* record number *)    1..100000000

p4 (* record area *)      area
```

This SSR serves to write the record specified by parameter 3 to a standard direct file previously opened as output or update file by SSR 4020. The file number (file_#) must specify the number

received from SSR 4020. The size of the area denoted by parameter 4 must be equal to the record length of the file.

If the file contains the records 1..n before this SSR is issued, then the record number specified by parameter 3 must be in the range 1..n for update files and in the range 1..n+1 for output files. A record can only be written to an update file if the immediately preceding SSR for this file has successfully read this record. It is not possible to read a record, unlock it (see SSR 4024), and then writing it back.

The completion code is returned to the PISA program and specifies:

- 0 = record successfully written
- 1 = record number outside allowed range (see above)
- 2 = block locked (see SSR 4022)
- 3 = write error, field "diagnostic" contains error message
- 4..10 not used by this SSR

If completion_code<>3, the value of the field "diagnostic" is undefined.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4024_____unlock_standard_direct_file

```
p2 record
    file_#:          1..1000
    completion_code:  0..10
    diagnostic:       string(30) vary
end record
```

This SSR serves to unlock the block of a standard direct file that has been previously locked by SSR 2022. If no block is locked by the program issuing this SSR, the "unlock" request is ignored and the completion code set to 1.

The completion code is returned to the PISA program and specifies:

- 0 = block successfully unlocked
- 1 = no block locked by this program
- 2 = unlock error, field "diagnostic" contains error message
- 3..10 not used by this SSR

If completion_code<>2, the value of the field "diagnostic" is undefined.

This SSR is only semiportable because the syntax of the error diagnostic is undefined.

4030-----delete_file

```
p2 record
    filename:          string(50) vary
    completion_code:   0..10
    diagnostic:        string(30) vary
end record
```

This SSR can be used to delete a file from secondary storage. It accepts the field "filename" and returns the fields "completion_code" and "diagnostic". A program must have write access to a file in order to delete it, otherwise a protection violation results.

filename: Indicates the file name under which the file to delete is known to the run-time system.

completion_code: Is returned to the PISA program and specifies:

- 0 = file deleted
- 1 = file not found
- 2 = file specified cannot be deleted
- 3 = protection violation
- 4 = file in use
- 5 = delete error, field "diagnostic" contains error message.
- 6..10 not used by this SSR

diagnostic: If completion_code=5, this field contains a diagnostic message, otherwise its value is undefined.

This SSR is only semiportable because the syntax of the fields "filename" and "diagnostic" is undefined.

4031_____rename_file

```
p2 record
    filename_old:      string(50) vary
    filename_new:      string(50) vary
    completion_code:   0..10
    diagnostic:        string(30) vary
end record
```

This SSR serves to give a new name to an existing file. A program must have write access to a file in order to rename it, otherwise a protection violation results. The "rename file" SSR accepts the fields "filename_old" and "filename_new" and returns the fields "completion_code" and "diagnostic".

filename_old: Gives the file name under which the existing file is known to the run-time system.

filename_new: Gives the new file name under which the file is known to the run-time system after successful completion of this SSR.

completion_code: Is returned to the PISA program and specifies:

```
0 = file renamed
1 = file not found
2 = file specified cannot be renamed
3 = protection violation
4 = file in use
5 = new file name already exists
6 = new file name is invalid
7 = rename error, field "diagnostic"
   contains error message.
8..10 not used by this SSR
```

diagnostic: If completion_code=7, this field contains a diagnostic message, otherwise its value is undefined.

This SSR is only semiportable because the syntax of the fields "filename_old", "filename_new", and "diagnostic" is undefined.

Semiportable system service requests (restricted)

No restricted semiportable SSRs are defined by now.

Semiportable system service requests (reserved)

6000 login

```
p2 record
    account:          string(30) vary
    password:         string(10) vary
    completion_code:  0..2
end record
```

The "login" SSR serves to switch a session from the login state to the attached state (cf. C-2.). The field "account" must specify the account number under which the session will be running; the field "password" has to contain the password for this account. The syntax of both these fields is implementation dependent.

The PISA run-time system returns a completion code to the program issuing this SSR:

```
completion_code:  0 = session switched to attached state
                  1 = invalid or nonexistent account
                  2 = invalid password
```

This SSR is only semiportable because the syntax of the fields "account" and "password" is implementation dependent.

Nonportable system service requests

All nonportable system service requests are defined in the Implementation Handbook of the PISA implementation introducing them.

This class of SSRs will particularly include SSRs for using special file access methods (index-sequential, library-oriented, etc.), for communicating with data base management systems, and for requesting implementation dependent services from the supporting run-time system.

If a PISA implementation allows the invocation of non-PISA procedures from a PISA program, the invocation must be done by a nonportable SSR. This SSR presumably requires the procedure name to be specified by an entire constant of type string as actual

parameter. The procedure could then be linked to the PISA program together with the external PISA procedures and functions (C-4.). The implementation handbook has to define the syntax of the non-PISA procedure names and the types of parameters accepted by such procedures.

Appendix B-III: Programming example

=====

This program reads an English text and produces a cross reference of the text's words. The words are kept in an AVL-balanced tree (see, e.g., [Wirth,76]). The line numbers of their occurrence are stored in linear lists linked to the corresponding tree element. (s. e. & o.)

main procedure crossref

```

type tree_elt =      record
    word:             string(16) vary
    first,last:       ->list_elt
    balance:          -2..+2  (* <0: left bias
                           =0: balanced
                           >0: right bias *)
    left,right:       ->tree_elt
end record

type list_elt =      record
    line:             1..10000
    succ:             ->list_elt
end record

type source_word = record
    word:             string(16) vary
    line:             1..10000
end record

var root,act_elt:    ->tree_elt

```

```
coroutine procedure nextword (var s: source_word)
```

```
(* Extract the next word from the input text and re- *)
(* turn it in "s.word" together with its line number *)
(* in "s.line". If the end of the text is reached, *)
(* return a nullstring in "s.word" *)
```

```
procedure open external (string(8), string(8), var 0..1000)
procedure get external (0..1000, area, var boolean)
procedure close external (0..1000)
```

```
var file: string(8)
var file_#: 0..1000
var line_#: 1..10000
```

```
loop open_file
  printnr ('file? ')
  input (file)
  open (file,'input',file_#)
  if file_#>0 then exit open_file end if
  print ('file ',file,' does not exist')
end loop open_file
```

```
loop read_input for line_#:=1 to 10000
```

```
  var line: string(80)
  var col: 1..80
  var eof: boolean
```

```
  get (file_#,line,eof)
  if eof then exit read_input end if
  s.word:=''
```

```
  loop scan_line for col:=1 to 80
```

```
    const alpha = set of char (:'A'..'Z','a'..'z','-':)
    detab
```

```
      cond line<:col,1:> in alpha "YNN"
```

```
      cond len(s.word)=0 "--YNN"
```

```
      cond col=80 "----Y"
```

```
      action "-X--"
```

```
        s.word:=s.word+cap (line<:col,1:>)
```

```
      action "----XX"
```

```
        s.line:=line_#
```

```
        suspend
```

```
        s.word:=''
```

```
    end detab
```

```
  end loop scan_line
```

```
end loop read_input
```

```
close (file_#)
```

```
end procedure nextword
```

```

procedure search(new_word: string(16) vary, var p: ->tree_elt,
                var grown: boolean)

  (* Search "new_word" in the tree with node "p->". If      *)
  (* found, return its address in "act_elt"; if not found, *)
  (* insert it at the proper place, keep the tree AVL-    *)
  (* balanced, and return its address in "act_elt".        *)

  procedure rotate (var root: ->tree_elt)
    var sub: ->tree_elt
    bind rb to root->.balance
    if rb<0 then      (* rotate with left successor *)
      bind sb to root->.left->.balance
      sub:=root->.left
      if sb<0 then rb:=rb-sb+1 else rb:=rb+1 end if
      sb:=sb+rb+1
      root->.left:=sub->.right; sub->.right:=root
    else      (* rotate with right successor *)
      bind sb to root->.right->.balance
      sub:=root->.right
      if sb>0 then rb:=rb-sb-1 else rb:=rb-1 end if
      sb:=sb-rb-1
      root->.right:=sub->.left; sub->.left:=root
    end if
    root:=sub
  end procedure rotate

  if p=nil then      (* insert new leaf *)
    new(p); grown:=true
    p->:=tree_elt(:new_word,nil,nil,0,nil,nil:)
    act_elt:=p; exit search
  end if

  if new_word<p->.word then
    search (new_word,p->.left,grown)
  if grown then      (* left subtree has grown *)
    p->.balance:=p->.balance-1
    if p->.balance<-1 then
      if p->.left->.balance>0 then
        rotate(p->.left)
      end if
      rotate (p)
    end if
    if p->.balance=0 then grown:=false end if
  end if
end if

```

```

else
  if new_word>p->.word then
    search(new_word,p->.right,grown)
    if grown then      (* right subtree has grown *)
      p->.balance:=p->.balance+1
      if p->.balance>+1 then
        if p->.right->.balance<0 then
          rotate(p->.right)
        end if
        rotate(p)
      end if
    end if
    if p->.balance=0 then grown:=false end if
  end if
else
  act_elt:=p
  grown:=false
end if
end if
end procedure search

```

```

coroutine function nextelt: ->tree_elt

```

```

(* Traverse the wordtree and return the addresses *)
(* of its nodes as function values (inorder).      *)
(* Return the value "Lil" after completion of      *)
(* the traversal.                                   *)

```

```

procedure traverse(p:->tree_elt)
  if p->.left<>nil then traverse(p->.left) end if
  nextelt:=p
  suspend
  if p->.right<>nil then traverse(p->.right) end if
end procedure traverse

```

```

traverse(root)
nextelt:=nil
end function nextelt

```

```

(* main program *)

root:=nil
loop build_xref
  var s:      source_word
  var grown:  boolean
  var p:      ->list_elt
  nextword(s)
  if s.word='' then exit build_xref end if
  search(s.word,root,grown)
  new(p)
  if act_elt->.last=nil then act_elt->.first:=p
  else act_elt->.last->.succ:=p end if
  act_elt->.last:=p
  p->:=list_elt(:s.line,nil:)
end loop build_xref

loop print_xref
  var tp: ->tree_elt
  var lp: ->list_elt
  tp:=nextelt
  if tp=nil then exit print_xref end if
  printnr(tp->.word,tab(20)); lp:=tp->.first
  loop print_line_# while p<>nil
    if col>75 then print(''); printnr(tab(20)) end if
    printnr(edit(lp->.line,'ZZZZ9 '))
    lp:=lp->.succ
  end loop print_line_#
  print(''); print('')
end loop print_xref

end procedure crossref

```

Appendix E-IV: Keyword index

=====

action entry	B-1.3.
action part	B-6.3.2.3.
activation of block	B-5.1.
actual parameter	B-4.3./B-6.2.2.
adaptable types	B-2.5.
area parameter	B-7./B-11.2.
array constant	B-3.1./B-3.2.
array type	B-2.3.1.
assignment statement	B-6.2.1.
associated type	B-2.4.
base type	B-2.3.3.
begin statement	B-6.3.1.
binding	B-3.5.
binding declaration	B-3.3.
block	F-5.
boolean type	B-2.2.1.2.
case statement	B-6.3.2.2.
char type	B-2.2.1.2.
comment	B-1.4.
compatibility (of types)	B-2.5.
compound statement	B-6.3.1.
condition entry	B-1.3.
condition part	B-6.3.2.3.
conditional statement	B-6.3.2.
constant	B-3.1./B-3.2.
constant declaration	B-3.1.
convertible types	B-2.5.
coroutine	B-7./B-8.
data type	B-2.
decision table statement	B-6.3.2.3.
declaration part	B-5.1.
detab statement	B-6.3.2.3.
digit	B-1.1.
discrete type	B-2.2.1.
dynamic variables	B-2.4.
entire constant	B-3.2.
entire variable	B-3.4.
enumerated type	E-2.2.1.1.
exit statement	B-6.2.4.
expression	B-4.

external function	B-8.2./B-9.2.
external procedure	B-7.2./B-9.2.
field	B-2.3.2.
field constant	B-3.2.
field identifier	B-2.3.2.
field variable	B-3.4.
fix(ed) type	B-2.2.2.
float type	E-2.2.2.
for construct	B-6.3.3.
formal parameter	B-7./B-8.
forward declarations	B-7.3./B-8.3.
function designator	B-4.3.
goto statement	B-6.2.3.
identifier	B-1.2.
if statement	B-6.3.2.1.
index	E-2.3.1.
indexed constant	E-3.2.
indexed variable	E-3.4.
integer	B-1.2.
integer type	B-2.2.1.2.
interfaces	B-11.
internal float	B-4.2.2.
irrelevant token	B-6.3.2.3.
label	B-6.1.
label declaration	E-6.1.
letter	E-1.1.
literal constant	B-3.2.
loop statement	E-6.3.3.
main procedure	E-9.
named block	B-5.1.
nested block	E-5.1.
nil	B-2.4.
no token	E-6.3.2.3.
nullstring	E-1.2.
number	E-1.2.
operator	B-4.2.
operator precedence	E-4.
packed	B-2.3.
parameter	B-4.3./B-6.2.2.
pointer type	B-2.4.
procedure declaration	B-7.

procedure statement	B-6.2.2.
program	B-9.1.
real type	B-2.2.2.
record constant	B-3.1./B-3.2.
record type	B-2.3.2.
record variable	B-3.4.
referenced variable	B-3.4.
repetitive statement	B-6.3.3.
root type	B-2.1.
rule (decision)	B-6.3.2.3.
rule token	B-1.3./B-6.3.2.3.
same type	B-2.5.
scope	B-5.2.
separator	B-1.4.
set constant	B-3.1./B-3.2.
set type	B-2.3.3.
set variable	B-3.4.
simple constant	B-3.1./B-3.2.
simple statement	B-6.2.
simple type	B-2.2.
simple variable	B-3.4.
SSR	B-11.2.
statement part	B-5.1.
static variables	B-2.4.
string	B-1.2.
string type	B-2.2.3.
structured constant	B-3.1./B-3.2.
structured type	B-2.3.
structured variable	B-3.4.
subrange type	B-2.2.1.3.
external routine	B-9.2.
substring constant	B-3.2.
substring variable	B-3.4.
suspension	B-7.1.
suspend statement	B-6.2.5.
sys procedure	B-11.2.
system interface	B-11.2.
system service request	B-11.2.
tag	B-2.3.2.
termination of block	B-5.1.
type	B-2.
type declaration	B-2.1.
type definition	B-2.
type identifier	B-2.1.
user interface	B-11.1.

value constructor	B-4.1.
value parameter	B-4.3./B-6.2.2.
variable	B-3.3./B-3.4.
variable declaration	B-3.3.
variable parameter	B-4.3./B-6.2.2.
variant	B-2.3.2.
vary option	B-2.2.3.
varying length string	B-2.2.3.
vocabulary	B-1.1.
while construct	E-6.3.3.
yes token	B-6.3.2.3.

Appendix B-V: References

=====

- [DEC1] Digital Equipment Corporation, "FSTS-E System User's Guide", Maynard, Massachusetts.
- [DEC2] Digital Equipment Corporation, "FSTS-E Programmer's Guide", Maynard, Massachusetts.
- [Haberman,73] Haberman A.N., "Critical Comments on the Programming Language Pascal", Acta Informatica 3, 1973, pp.47-57.
- [Knuth,68] Knuth D.E., "The Art of Computer Programming", Vol.1, Addison-Wesley, Reading, 1968, pp.190-196.
- [Lampson et al.,77] Lampson B.W., Horning J.J., London R.L., Mitchell J.G., and Popek G.L., "Report on the Programming Language Euclid", Sigplan Notices, Vol.12, No.2, Feb. 1977.
- [Peterson et al.,73] Peterson W.W., Kasani T., and Tokura N., "On the Capabilities of While, Repeat, and Exit Statements", CACM, Vol.16, No.8, 1973, pp.503-512.
- [Pollack et al.,71] Pollack S.L., Hicks H.T., Jr., and Harrison W.J., "Decision Tables: Theory and Practice", Wiley & Sons, New York, 1971.
- [Ritchie/Thompson,74] Ritchie D.M. and Thompson K., "The UNIX Time-Sharing System", CACM, Vol.17, No.7, 1974, pp.365-375.
- [Thurner,71] Thurner R., "Entscheidungs-Tabellen: Aufbau-Anwendung-Programmierung", VDI-Taschenbuch T33, VDI Verlag, Duesseldorf, 1971.
- [Thurner,73] Thurner R., "Detab-GT-User Manual Cobol", published by the author, Duebendorf (Switzerland), 1973.

- [Welsh et al.,77] Welsh J., Sneeringer W.J., and Hoare C.A.R., "Ambiguities and Insecurities in Pascal", Dept. of Computer Science, Queen's University, Belfast, 1977.
- [Wirth,75] Wirth N., "Pascal Report", in: Jensen K. and Wirth N., "PASCAL - User Manual and Report", second edition, Springer-Verlag, New York, 1975, pp.133-167.
- [Wirth,76] Wirth N., "Algorithms + Data Structures = Programs", Prentice-Hall, Englewood Cliffs, N.J., 1976, pp.216-222.
- [Wortman,77] Wortman D.B., "On Legality Assertions in Euclid", Computer Systems Research Group, University of Toronto, 1977.

SECTION C

The PISA Session

The PISA system serves to create, test, use, and maintain application software based on PISA programs. Each of these activities involves a dialogue between the user and the PISA system. The user is either a person occupied with software production, a person interacting with application programs, or a PISA program that makes use of the PISA system to process specific tasks. Persons maintain their dialogue with the PISA system via a real terminal while programs employ a pseudo terminal. At the moment a user establishes connection to the PISA system, a PISA session is assigned to him; the PISA session provides the user with the hardware and software resources supported by the PISA system.

This section defines the basic concepts of a PISA session, the states of the session, and its modes. The states of a session are introduced in C-2.; they represent the condition of the connection between the user and the PISA system. Chapters C-3. to C-7. describe the modes of a PISA session. The mode of a session indicates the kind of activity the user is currently performing:

Mode	Activity
-----	-----
system	select a program to execute
edit	prepare PISA program text for execution
run	execute a PISA program in C-code (slow code)
execute	execute a PISA program in T-code (fast code)
halt	investigate a program malfunction

Each mode defines its dialogue and the conditions for switching the session to another mode. The mode transitions are summarized in the graph shown in Appendix C-II.

Table of contents

NOTATION AND BASIC CONCEPTS	C-1.
Notation	C-1.1.
Terminal dialogue	C-1.2.
Message packing	C-1.2.1.
Deleting input text	C-1.2.2.
Stalling the output	C-1.2.3.
User interrupts	C-1.2.4.
Standard prompter	C-1.2.5.
Format of error messages	C-1.2.6.
Accounts and privileges	C-1.3.
Files	C-1.4.
Libraries, members, and programs	C-1.5.
THE STATES OF A PISA SESSION	C-2.
The idle state	C-2.1.
The login state	C-2.2.
The attached state	C-2.3.
The detached state	C-2.4.
THE SYSTEM MODE	C-3.
The basic structure of the system mode	C-3.1.
Standard system programs	C-3.2.
The "login" system program	C-3.2.1.
The "logout" system program	C-3.2.2.
The "attach" system program	C-3.2.3.
The "term" system program	C-3.2.4.
Standard utility programs	C-3.3.
THE EDIT MODE	C-4.
The basic structure of the edit mode	C-4.1.
Switching the session to system mode	C-4.1.1.
The edited text	C-4.2.
Text lines	C-4.2.1.
List directive statements	C-4.2.2.
PISA program text	C-4.2.3.
Ranges of text lines	C-4.2.4.
Modifying the edited text	C-4.3.
Adding a single text line	C-4.3.1.
Adding a sequence of text lines	C-4.3.2.
Including text lines	C-4.3.3.
Deleting text lines	C-4.3.4.
Changing text lines	C-4.3.5.
Renumbering text lines	C-4.3.6.

Inspecting the edited text	C-4.4.
Printing a range of text lines	C-4.4.1.
Printing selected text lines	C-4.4.2.
Producing a program listing	C-4.5.
Saving the edited text	C-4.6.
Compiling the edited text	C-4.7.
Translating the edited text	C-4.8.
Checking the edited text	C-4.9.
Running the edited text	C-4.10.
 THE RUN MODE	 C-5.
The basic structure of the run mode	C-5.1.
The C-code	C-5.2.
Run-time error messages	C-5.3.
Interruption locations	C-5.4.
Run-time errors	C-5.4.1.
Expression evaluation errors	C-5.4.1.1.
Constant and variable denotation errors	C-5.4.1.2.
Procedure and function seizing errors	C-5.4.1.3.
Errors in predefined procedures	C-5.4.1.4.
Predefined procedure "stop"	C-5.4.2.
User interrupt	C-5.4.3.
Block termination	C-5.4.4.
 THE EXECUTE MODE	 C-6.
The basic structure of the execute mode	C-6.1.
The T-code	C-6.2.
Execution error messages	C-6.3.
 THE HALT MODE	 C-7.
The basic structure of the halt mode	C-7.1.
Halt commands	C-7.2.
Switching the session to system mode	C-7.2.1.
Switching the session to edit mode	C-7.2.2.
Continuing execution of interrupted program	C-7.2.3.
Requesting the interruption location	C-7.2.4.
Immediate PISA statements	C-7.3.
 Appendix C-I: Session state transition graph	
 Appendix C-II: Session mode transition graph	
 Appendix C-III: Sample session	
 Appendix C-IV: Compile-time errors	

Appendix C-V: Run-time errors

Appendix C-VI: Collected PISA command syntax

Appendix C-VII: Keyword index

C-1. NOTATION AND BASIC CONCEPTS

C-1.1. Notation

The syntax of many constructs in the session's dialogue is described in slightly modified Backus-Naur form. It is the same dialect of BNF that has been used in Section B to describe the syntax of the programming language PISA. The dialect has been introduced in the first paragraph of B-1.1. An end-of-line character of a text that is input on the terminal is represented by the fat dot "." throughout this section.

All terminal symbols in the syntax definitions are written in small letters. The implementation is obliged to correctly process a dialogue even if some or all of the symbol's characters are capital letters, i.e. a capital letter within a symbol has to be syntactically treated as a small letter.

Whenever PISA program text is used to describe the syntax and semantics of a dialogue, its meaning is based on the PISA language definitions given in Section B (including the predefined procedures and functions). Such a PISA text is by no means intended to textually define the implementation of the dialogue; it merely serves as a precise "meta language" to describe the dialogue.

The user of a PISA session is either a person operating a terminal connected to PISA or a program controlling the session via a pseudo terminal. Pseudo terminal operations have been introduced with the system service requests 2000 to 2004 in Appendix B-II. It is of no concern to a PISA session whether it is connected to a real terminal or to a pseudo terminal. To simplify terminology, the explanations in this section assume a connection to a real terminal; all definitions, however, hold for pseudo terminal operation accordingly.

C-1.2. Terminal dialogue

The user interacts with PISA via a terminal. As the following chapters show, the terminal dialogue is processed either by an executing PISA program or by the PISA run-time system. The processing software defines the syntax and the semantics of the

dialogue. Regardless of the software processing a dialogue, however, the basic properties of a terminal dialogue described below hold for any interaction between the terminal and PISA.

C-1.2.1. Message packing

Text entered at the terminal is not passed to the PISA program or PISA run-time system as long as it is not terminated by an end-of-line character. The "carriage-return", "line-feed", and "escape" characters act as end-of-line characters. All text entered up to and including the end-of-line is called an input line. An input line may not exceed 255 characters (including the end-of-line character). This holds independent of whether the terminal is a real or a pseudo terminal. Input lines can be entered in advance, i.e. before they are requested by the software controlling the terminal dialogue. The PISA terminal handler buffers them. If the buffer is full, further input is inhibited or ignored.

Text produced for output on the terminal is stored in a buffer by the PISA run-time system. The PISA terminal handler constantly transfers the contents of the buffer to the terminal's output device, i.e. "carriage-return", "line-feed", and "escape" characters have no special packing function within output to the terminal. If the output buffer is full, the session is automatically stalled until the buffer is able to accept further text.

Chapter E-3. gives more details about the PISA terminal handler. For the common user of PISA, it is sufficient to know that input data is packed into lines while output data is written on the terminal character by character, as it is produced.

C-1.2.2. Deleting input text

Text on the actual (incomplete) input line can be deleted as long as the text is not yet passed to the PISA program or the PISA run-time system. After the actual input line has been terminated by an end-of-line, its text can no longer be modified.

A backspace character causes the deletion of the rightmost character on the actual input line and backspacing the cursor or write mechanism of the terminal by one character. If the

terminal is a scope terminal operating in full-duplex mode, the backspace character entered results in a backspace/space/backspace echo to the terminal. This visually erases the rightmost character of the input text. If the input text is empty, all backspace characters are ignored.

The entire actual input line may be deleted by entering a delete line character. The PISA terminal handler prints the text "!<" behind the last character of the input line to mark the line as deleted. Then, the cursor or write mechanism is moved to column 1 of the next line. This happens regardless of whether the actual input line is empty or not. The implementation defines the "delete line" character; for ASCII terminals, the character NAK (CNTL/U) should be used whenever feasible.

C-1.2.3. Stalling the output

As stated in C-1.2.1., the PISA terminal handler continuously writes the output from the terminal output buffer to the terminal. This operation can be stalled at any time by entering a stall output character at the terminal; the terminal handler resumes the output as soon as a resume output character is entered. Stalling the output does not affect the echoing of input on a terminal operating in full-duplex mode. The implementation defines the "stall output" and "resume output" characters; for ASCII terminals, the following characters should be used whenever feasible: DC3 (CNTL/S) for "stall output" and DC4 (CNTL/T) for "resume output".

Stalling of output must not necessarily be supported for terminals operating in half-duplex mode.

C-1.2.4. User interrupts

A user interrupt is signalled by entering a user interrupt character. The implementation defines this character; for ASCII terminals operating in full-duplex mode, the character ETX (CNTL/C) should be used whenever feasible. When data is transmitted in half-duplex mode, the BREAK (or ATTN) key must be used to signal a user interrupt.

User interrupts serve to discontinue the execution of the active program. Every program may disable and enable the user interrupt

facility (cf. B-11.1.). All user interrupts entered during the time the user interrupt facility is disabled are ignored.

The effect of an enabled user interrupt depends on the mode of the session. It will be described for every mode in chapters C-3. to C-7. Regardless of the mode-dependent actions caused by an enabled user interrupt, it always

1. deletes all data from the terminal's input and output buffers, i.e., the input that has not yet been processed and the output that has not yet been written to the terminal,
2. sets the echo flag to "on" (cf. B-11.1.),
3. ensures that further output is no longer stalled (cf. C-1.2.3.), and
4. terminates the stalling of program execution if the program is currently stalled.

C-1.2.5. Standard prompter

Whenever the PISA run-time system does not use a text to indicate to the user that it waits for an input from the terminal, it prints a standard prompter. A standard prompter consists of the sequence underscore/backspace, i.e., the first character entered overprints the underscore.

C-1.2.6. Format of error messages

All error messages issued by any component of the PISA system have the following format:

```
error_msg    ::= "*" type number "*" " text
type         ::= "S" | "E" | "R" | "X" | "H" | "P"
number       ::= digit digit digit
text         ::= {character}
```

Examples: *S012* program not found
 E141 line truncated
 X037 at line 1260 in join
 X158

The type character indicates the type of the error:

S = system mode errors and errors that cannot be
 associated with a specific mode
 E = edit mode errors
 R = run-time errors (detected in run mode)
 X = execution errors (detected in execute mode)
 H = halt mode errors
 P = programming errors detected during compilation or
 translation of PISA program or external routine

Every error has a unique error number. Some error numbers are implementation dependent while others are the same on all PISA implementations. The numbers of all errors in the latter category are defined in this section. The error number 999 is used to report an abnormal malfunction of the PISA system (see Chapters C-2. to C-7.).

The error text is always implementation and even installation dependent. All error texts given in this report are just English examples of appropriate error texts. These error texts may be replaced by any texts in any language. Programs controlling a PISA session via a pseudo terminal should use only the error type and the error number to identify errors reported by the controlled session.

C-1.3. Accounts and privileges

The textual definition of user account identifications, their privileges and associated priorities, and the relation between the file system and the user accounts is dictated by the operating system on which a PISA implementation is based. Therefore, a PISA session relies only on a very few properties of the account and privilege scheme:

- (1) Every user must enter his account identification at the time he logs in to PISA. The syntax of the account identification is implementation dependent. The implementation also defines whether or not more than one

user with the same account can use the PISA system simultaneously.

- (2) Every account falls into one of the three PISA privilege classes "no privileges", "restricted privileges", and "full privileges". The implementation defines the mapping of the actual privilege scheme to the PISA privilege classes (in the worst case, all accounts have full privileges). The only instances where PISA uses the privilege information is in the "get session parameters" SSR (cf. Appendix B-II) and to decide whether or not a user is allowed to compile or translate certain SSRs (cf. B-11.2.).

C-1.4. Files

The syntax of filenames and the protection mechanism imposed on file accesses is completely undefined in PISA. All system service requests that access files are semiportable or non-portable (cf. E-11.2. and Appendix B-II).

C-1.5. Libraries, members and programs

Although the entire file system is implementation dependent, the following properties of libraries, members, and programs are a minimal requirement for every PISA implementation:

- (1) Every account has three libraries associated with it: they are called "user S-library", "user C-library", and "user T-library". The S-library holds PISA source text, the C-library holds programs and external routines in C-code (C-5.), the T-library holds programs and external routines in T-code (C-6.).
- (2) Besides its "user" S-library, C-library, and T-library, every account has the "project" S-library, C-library, and T-library associated with it. An account belongs to one and only one project. The relation between an account and a project is defined by the implementation; it can be an implicit relation given by the account identification, or it can be defined by explicit relations.

- (3) The "system" S-library, C-library, and T-library are common to all accounts in all projects. Each of the three system libraries exists only once on the PISA system.
- (4) Every user has access to its "user" libraries, to the associated "project" libraries, as well as to the "system" libraries. The access is granted independent of any privileges and protection codes.
- (5) Each library contains zero or more members. The syntax of the member names is implementation dependent. Since members of C-libraries and T-libraries get their names from PISA procedure identifiers (cf. C-4.7. and C-4.8.), the syntax of their member names must be the same as the syntax of PISA identifiers or a restriction thereof.
- (6) A member of a C-library or T-library is called a "program" if it contains a PISA main procedure (B-9.1.).

If the operating system on which the PISA implementation is based does not support library files, libraries must be simulated in some way to comply with the above requirements.

C-2. THE STATES OF A PISA SESSION

=====

A PISA session either has a terminal physically connected to it, or it has no connection to a terminal. On the other hand, a PISA session is either logged in to the PISA system, or it is not logged in (or logged out). The four combinations of these conditions determine the four possible states of a session: "idle", "login", "attached", and "detached". This chapter defines the four states and gives the conditions for the state transitions. Every session in a PISA system has a unique session number. The session number is assigned at the run-time system's discretion and is in the subrange 1..1000 of integer.

C-2.1. The idle state

A PISA session in the idle state has no terminal connected to it and is logged out from the PISA system. It has no programs running in it and does not wait for data input. Usually, no process exists in the PISA run-time system for an idle session. Therefore, such a session does not occupy any resources of the host computer, except, perhaps, for some small table entries in the PISA nucleus. All sessions of a PISA programming system are initially in the idle state.

Whenever a physical connection is set up between a terminal and PISA, the PISA run-time system assigns an idle session to the terminal and switches this session from the idle state to the login state.

A real terminal is physically connected to PISA whenever a bidirectional data path exists between the real terminal and PISA that is able to transfer data at a defined rate between the two. Depending on the supporting hardware and system software, such a physical connection is either established as soon as the terminal is connected to PISA or at the time the terminal transmits the first data.

A pseudo terminal is physically connected to PISA after successful completion of the "create pseudo terminal" system service request (SSR 2000).

There are certain conditions in a PISA system which do not permit to create a session in the login state for a connected terminal.

The following error messages are written on the terminal in the case where they apply:

```
*S001* PISA not active -- try later
      There is no full PISA system running on the host
      computer.

*S002* maximum number of PISA sessions active -- try later
      There are no idle sessions available.

*S003* no logins to PISA accepted now -- try later
      The operator has disabled the logins to PISA.

*S999* PISA system error
      The PISA run-time system encountered some error
      while trying to switch the session from the idle
      state to the login state. This message should be
      followed by one ore more lines of information
      helping to trace the error.
```

The implementation handbook of a PISA implementation defines the default characteristics assumed for a terminal connected to PISA. These default characteristics can be changed with the system service request 1006.

C-2.2. The login state

A PISA session in the login state has a terminal physically connected to it, but is not logged in to the PISA system. The session is not operating under any account and has no privileges. The main purpose of the login state is to provide the environment for running the "login" program as described in C-3.1. It, therefore, can be viewed as an intermediate state, initiated after a physical connection has been set up and finally leading to the attached state after a successful login (or, in other words, after a successful "logical" connection).

A session is switched from the login state to the attached state by issuing a successful "login" system service request (SSF 6000).

When the physical connection between the terminal and a session in login state is disrupted, the PISA run-time system switches the session from the login state to the idle state. A connection to a real terminal is considered to be disrupted when the

conditions for its physical connection are no longer fulfilled (see C-2.1.). The connection to a pseudo terminal is disrupted after the controlling program issued a "delete pseudo terminal" system service request for this pseudo terminal (SSR 2001).

C-2.3. The attached state

The attached state is the common state of a PISA session that maintains a dialogue with its user. An attached session is physically connected with a terminal, is logged in, and has an account assigned to it. It is the account specified by the "login" system service request that switched this session from the login state to the attached state.

A session can be switched from the attached state to the login state by issuing a successful "logout" system service request (SSR 3000).

When a session is used to run a program or a batch of programs that do not require any data exchange with the terminal, the session may be disconnected from the terminal. This is accomplished by executing a "detach" system service request (SSR 3001), which switches a session from the attached state to the detached state.

If the physical connection between the terminal and a session in attached state is disrupted, the PISA run-time system switches the session from the attached state to the detached state. Disrupted connections to real and pseudo terminals have been described in C-2.2.

C-2.4. The detached state

PISA sessions in the detached state are commonly used to run long programs or batches of programs that neither transfer any data to a terminal nor expect any input therefrom. Besides such applications, the detached state is also used to catch sessions that lost the connection to their terminal (see C-2.3.).

A detached session has no terminal connected to it but is logged in and runs under the account assigned to the session at login. Any transfer of data to the user interface of a detached session immediately stalls execution of the session until it is

reattached. This occurs regardless of whether the data transfer was from the PISA run-time system or a program running in the session. Similarly, a detached session waiting for terminal input is stalled until it gets the input, and this is only possible in the attached state.

A successful "logout" system service request (SSR 3000) issued in a detached session switches this session from the detached state to the idle state.

A session can be switched from the detached state to the attached state by issuing a successful "attach" system service request (SSR 3002). The SSR must be issued by a program running in the attached state under the same account as the detached session.

C-3. THE SYSTEM MODE

=====

After a session has been switched from the idle state to the login state, the session is in system mode. The system mode allows a user to run any accessible program and provides a command to switch the session from system mode to edit mode. This chapter defines the basic structure of the system mode and introduces the so called "standard system programs" and "standard utility programs".

C-3.1. The basic structure of the system mode

As long as the PISA session is in system mode, the terminal is controlled by the PISA run-time system. The basic structure of the system mode is defined by the PISA program below.

```
main procedure system_mode

var p_4002: record
    session_no:    1..1000
    priv:          0..2
    state:         0..2
    line:          string(10) vary
end record
var comm_string:  string(255) vary
const backspace = 8  (* ASCII BS *)

interrupt(true)
sys(4002,p_4002)  (* get session parameters *)
```

```

if (* previous program chained to another program *)
  (* by SSR 4000 (cf. Appendix E-II) *) then

  (* Search the program first on the user T- and *)
  (* C-libraries, then on the project T- and C-lib- *)
  (* raries, and finally on the system T- and C-lib- *)
  (* raries. On each level, the T-library is searched *)
  (* before the C-library. If the session is in login *)
  (* state, only the system libraries are searched. *)
  (* If the program has been found and is valid, it *)
  (* is loaded into the program area and the session *)
  (* switched to run mode or execute mode. Run mode *)
  (* is selected for C-code, execute mode for T-code. *)
  (* If the program has not been loaded successfully, *)
  (* execution continues below. *)

  print('S010* cannot chain to program <m>')
    (* <m> is replaced by the program name *)

end if

if p_4002.state = 0 then (* session is in login state *)
  comm_string := ''

  (* If the program "login" exists on the system T- *)
  (* library, the T-code is loaded into the program *)
  (* area and the session switched to execute mode, *)
  (* else if it exists on the system C-library, the C- *)
  (* code is loaded and the session switched to run *)
  (* mode, else execution continues below. *)

  print ('S011* login program not found')
  loop wait_forever (* infinite loop until connection is *)
    sleep(60) (* disrupted or user interrupt *)
  end loop wait_forever (* occurs *)

end if

```

```

print('system mode')
loop get_input
  var line:                string(255) vary
  var delim:               0..255
  var program:             string(16) vary

  printnr('_',val(char,backspace))  (* prompter *)
  input(line)

  if small(line) = 'edit' then
    (* switch session to edit mode; execution *)
    (* continues as described in C-4.          *)
  end if

  delim := index(line,' ')
  if delim = 0 then
    program := line
    comm_string := ''
  else
    program := line<:1,delim-1:>
    comm_string := line<:delim+1:>
  end if

  (* The program is searched first on the user T- and      *)
  (* C-libraries, then on the project T- and C-libraries, *)
  (* and finally on the system T- and C-libraries. On     *)
  (* each level, the T-library is searched before the     *)
  (* C-library. If the program has been found and is      *)
  (* valid, it is loaded into the program area and the    *)
  (* session switched to run or execute mode. Run mode    *)
  (* is selected for C-code, execute mode for T-code.     *)
  (* If the program has not been loaded successfully,     *)
  (* execution continues below.                            *)

  print('*S012* program not found')
end loop get_input

end procedure system_mode

```

Notice that the procedure controlling a session in system mode, in whatever programming language it is written, is not performed in run or execute mode, but always in system mode.

The basic structure of the system mode shows that the user interrupt facility is enabled each time a session is switched to system mode. A user interrupt during system mode

1. discontinues the current activity and does the necessary clean-up and
2. passes control to the first statement in the procedure "system_mode".

Discontinuing the current activity never leaves an illegal state on any PISA system component. The PISA run-time system may disable the user interrupt facility during critical sections in system mode in order to fulfill this requirement.

If any abnormal error occurs in system mode, the PISA run-time system writes the error message

S999 PISA system error

on the terminal. This message should be followed by one or more lines of information that help tracing the error. After the messages have been produced, the implementation is free to either reestablish the session in system mode or to cancel the session and switch it to the idle state. The first version is preferred.

As the description of the system mode exhibits, the system mode distinguishes essentially between three cases when it is entered:

- (1) If the session is in the login state, the system program "login" is given control regardless of the terminal input. It is the task of this program to process the dialogue the installation allows in login state. This unconditional activation of the "login" program makes sure that a prescribed login dialogue cannot be eluded.
- (2) If the session is in the attached state and the input text is the keyword "edit", the session is switched to edit mode.
- (3) If the session is in the attached state and the input text is not the keyword "edit", the text entered is assumed to specify the name of the program to be executed, optionally followed by the communication string to be passed to this program. All characters preceding the first blank of the input text are interpreted as the program name, all characters succeeding the first blank as the communication string. If the text contains no blank, the communication string is a nullstring.

Examples:

```

copy file3 to prt•   program='copy'
                    comm_string='file3 to prt'

custlist•           program='custlist'
                    comm_string=''

```

C-3.2. Standard system programs

The programs described in the sequel are called "standard system programs" because they are provided on every PISA installation. They are available in source form so that they can be modified to meet the requirements of an individual installation. But, however they are changed, they have to support the dialogue as defined by the basic versions of the system programs given below.

C-3.2.1. The "login" system program

The "login" program serves to switch a PISA session from the login state to the attached state.

main procedure login

```

var p_4002 record
    session_no:    1..1000
    priv:          0..2
    state:         0..2
    line:          string(10) vary
end record
var p_6000: record
    account:       string(30) vary
    password:      string(10) vary
    compl_code:    0..2
end record
var line:         string(255) vary

interrupt(true)
sys(4002,p_4002)  (* get session parameters *)

```



```

if p_4002.state <> 0 then  (* session not in login state *)
  print('*S101* session already logged in')
  exit login
end if
input(line)
if small(line) = 'login' then
  printnr('PISA account? ')
  input(p_6000.account)
  printnr('password? ')
  echo(false)
  input(p_6000.password)
  echo(true)
  sys(6000,p_6000)  (* login *)
  case p_6000.compl_code
    when 0:
      when 1: print('*S102* invalid account -- login rejected')
      when 2: print('*S103* invalid password -- login rejected')
    end case
else
  print('*S100* invalid login to PISA')
end if
end procedure login

```

C-3.2.2. The "logout" system program

The "logout" system program serves to switch an attached PISA session to the login state or a detached session to the idle state (cf. C-2.).

```

main procedure logout

var p_4001: record
  acct:          string(30) vary
  el,cpu,lo,io:  -1..100000000
end record

var p_4002: record
  session_no:    1..1000
  priv:          0..2
  state:         0..2
  line:          string(10) vary
end record

```

```

sys(4002,p_4002)  (* get session parameters *)
if p_4002.state = 0 then (*session is in login state*)
    print('*S110* session already logged out')
    exit logout
end if
interrupt(false)
if p_4002.state = 1 then (*session is in attached state*)
    sys(4001,p_4001)  (* get accounting data *)
    if p_4001.el <> -1 then
        print('elapsed time: ', p_4001.el div 6000, ' min ',
              p_4001.el mod 6000 div 100, ' sec')
    end if
    if p_4001.cpu <> -1 then
        print('cpu time: ', p_4001.cpu div 6000, ' min ',
              p_4001.cpu mod 6000 div 100, ' sec')
    end if
    if p_4001.io <> -1 then
        print('i/o activity: ', p_4001.io, ' transfers')
    end if
    print('session logging out')
end if
sys(3000)  (* logout *)

end procedure logout

```

Notice that the "logout" system program does not print messages to the terminal if the session is in the detached state. This prevents a detached session chaining to the logout program from being stalled because of a data transfer to the terminal (see C-2.4.).

The "logout" system program will probably be modified in practice to update accounting data before the session is logged off.

C-3.2.3. The "attach" system program

The "attach" system program is used to attach a detached session to the terminal.

main procedure attach

```

type session_# =                1..1000
var p_3002: record
    session_no: session_#
    comp_code: 1..3
end record
var p_4003: record
    cnt1: 1..3
    session_no: session_#
    state: 0..2
    mode: 0..4
    line: string(10) vary
    program: string(16) vary
end record
var line: string(255) vary
const state = array (0..2) of string(5)
               (: 'log', 'att', 'det':)
const mode = array (0..4) of string(5)
               (: 'sys', 'edit', 'halt', 'run', 'exec':)

interrupt(true)
loop get_session_#
    printnr('session to attach to? ')
    input(line)
    if line = '?' then
        p_4003.cnt1 := 1; sys(4003,p_4003)
        print(' no state mode line program')
        loop session_info while p_4003.cnt1<>3
            print(edit(p_4003.session_no,'ZZZ9 '),
                  state(p_4003.state), mode(p_4003.mode),
                  p_4003.line, tab(30), p_4003.program)
            sys(4003,p_4003)
        end loop session_info
        print('')
    else

```

```

if tconv(session_#,line) then
  p_3002.session_no := conv(session_#,line)
  sys(3002,p_3002)  (* attach *)
  case p_3002.comp_code
    when 1: print('*S120* attach SSR in nonattached ',
                  'session.')
    when 2: print('*S121* this session is not active ',
                  'under your account')
    when 3: print('*S122* this session is not detached')
  end case
else
  print('*S123* invalid session number')
end if
end loop get_session_#
end procedure attach

```

C-3.2.4. The "term" system program

When a physical connection is set up between a terminal and PISA, the PISA run-time system assumes certain default characteristics for the terminal. Every PISA implementation defines these default characteristics. If they do not match the characteristics of the terminal connected, the "term" system program can be used to change the default characteristics to the actual ones.

The dialogue of the "term" system program may be adapted to the needs of a PISA implementation. Therefore, no sample "term" program is given here. The lack of an exact definition of the "term" dialogue does not affect the portability of programs controlling pseudo terminals since the pseudo terminal's default characteristics are virtually never changed.

C-3.3. Standard utility programs

Besides the standard system programs introduced in C-3.2., every PISA implementation has to provide utility programs to accomplish the following functions:

- deleting files, libraries, and library members
- renaming files, libraries, and library members
- copying files, libraries, and library members
- printing directories of files and libraries

All these functions must be executable from the system mode in a similar way as defined for the standard system programs.

The dialogue of the standard utility programs is not defined. This has two main reasons:

- (1) Most operating systems already provide utility programs which are adapted to the needs of the implementation.
- (2) Program translators and program preprocessors controlling a PISA session via a pseudo terminal rely heavily on the dialogue of the controlled session. Since these programs will probably not use utilities, their portability is preserved even without defining utility dialogues.

C-4. THE EDIT MODE

=====

The edit mode provides a text area wherein a PISA program or external routine can be prepared for execution. The text held in the text area is called "edited text". The edit mode includes commands to modify and inspect the edited text, to save it for later usage, to check it for syntactic correctness, and to transform it into executable code. A "list" command is available to request a formatted source listing of the PISA program or external routine in the edited text. If the edited text holds a PISA main procedure (B-9.1.), it can be run with the "run" command, which compiles the text and switches the session to run mode. The "sys" command gives the possibility of switching the session back to system mode.

This chapter defines the dialogue of the edit mode. A PISA implementation is free to introduce additional edit commands. The dialogue introduced below, however, must be supported on every PISA implementation.

C-4.1. The basic structure of the edit mode

As long as the PISA session is in edit mode, the terminal is controlled by the PISA run-time system. The basic structure of the edit mode is defined by the PISA program below.

```
main procedure edit_mode
```

```
const backspace =      8 (* ASCII BS *)
const numeric =      set of char (:'0'..'9':)
var line:            string(255) vary
var command:         string(10) vary
```

```
interrupt(true)
print('edit mode')
```

```
*> next_command:
```

```
printnr('_',val(char,backspace))          (* prompter *)
input(line)
command := small(line<:1,index(line,' '):>)
if command='' then command:=small(line) end if
```

```

if len(line) < 2 then
  print('*E001* invalid edit command')
  goto next_command
end if

if command = 'sys' then
  (* switch session to system mode; execu- *)
  (* tion continues as described in *)
  (* C-3.1. *)
end if

if command<:1,1:> in numeric then
  (* add single text line, cf. C-4.3.1. *)
  goto next_command
end if

if command = 'a' then
  var line_#, increment: 1..999999
  (* decode add command, cf. C-4.3.2. *)
  (* set line_# and increment *)
  loop generate_line_#
    printnr(line_#, tab(8))
    input(line)
    if small(line)='e' and len(line)=1 then
      goto next_command
    end if
    line := str(line_#) + ' ' + line
    (* add single text line, cf. C-4.3.1. *)
    if line_# > 999999-increment then
      goto next_command
    end if
    line_# := line_# + increment
  end loop generate_line_#
end if

if command = 'i' then
  (* decode and execute include *)
  (* command, cf. C-4.3.3. *)
  goto next_command
end if

if command = 'd' then
  (* decode and execute delete *)
  (* command, cf. C-4.3.4. *)
  goto next_command
end if

```

```

if command = 'compile' then                                (* compile *)
  (* decode and execute compile *)
  (* command, cf. C-4.7. *)
  goto next_command
end if

if command = 'check' then                                  (* check *)
  (* decode and execute check *)
  (* command, cf. C-4.9. *)
  goto next_command
end if

if command<:1,1:> = 'c' then                                (* change *)
  (* decode and execute change *)
  (* command, cf. C-4.3.5. *)
  goto next_command
end if

if command = 'run' then                                    (* run *)
  (* decode and execute the run *)
  (* command, cf. C-4.10. *)
  end if

if command = 'r' then                                      (* renumber *)
  (* decode and execute renumber *)
  (* command, cf. C-4.3.6. *)
  goto next_command
end if

if command = 'p' then                                      (* print *)
  (* decode and execute print *)
  (* command, cf. C-4.4.1. *)
  goto next_command
end if

if command = 'v' then                                      (* verify *)
  (* decode and execute verify *)
  (* command, cf. C-4.4.2. *)
  goto next_command
end if

if command = 'list' then                                   (* list *)
  (* decode and execute list *)
  (* command, cf. C-4.5. *)
  goto next_command
end if

```



```

if command = 'save' then
    (* decode and execute save *)
    (* command, cf. C-4.6. *)
    goto next_command
end if

if command = 'translate' then
    (* decode and execute translate *)
    (* command, cf. C-4.8. *)
    goto next_command
end if

print('*E001* invalid edit command')
goto next_command

end procedure edit_mode

```

If an error is detected during decoding and executing an edit command, the appropriate error message is written on the terminal and control branched to the label "next_command". The error messages are defined below together with the edit commands.

As the description of the edit mode's basic structure shows, the user interrupt facility is enabled each time a session is switched to edit mode. A user interrupt during edit mode

- (1) discontinues the current activity and does the necessary clean-up and
- (2) passes control to the first statement in the procedure "edit_mode".

Discontinuing the current activity never leaves an illegal state on any PISA system component. The PISA run-time system may disable the user interrupt facility during critical sections in edit mode in order to fulfill this requirement.

If any abnormal error occurs in edit mode, the PISA run-time system writes the error message

E999 PISA system error

on the terminal. This message should be followed by one or more lines of information that help tracing the error. After the messages have been produced, the implementation is free to either reestablish the session in edit mode or system mode or to cancel the session and switch it to the idle state. The preferred version is to reestablish it in edit mode without loss of data.

C-4.1.1. Switching the session to system mode

The sys command allows to switch the session from edit mode to system mode.

```
sys_cmd      ::= "sys•"
```

If the edited text has been modified since processing the last save command (C-4.6.), the user must answer the message

really switch to system mode without saving?

with any string starting with a "y" or "Y". Otherwise, the session mode is not switched to system mode but control passed to the label "next_command" (cf. C-4.1.).

C-4.2. The edited text

The edited text consists of 0 to n text lines:

```
edited_text  ::= {text_line}
```

Initially, i.e. when the session is switched from system mode to edit mode, the edited text does not contain any text lines.

C-4.2.1. Text lines

A text line can be subdivided into a line number, followed by a separator, followed by either a list directive statement or PISA program text.

```
text_line    ::= line_number sep (list_dir_stmt | pisa_prog_text)
line_number  ::= digit {digit}
sep          ::= " " {" "}
```

The line number contains 1 to 6 digits and is greater than zero. It never contains leading zeroes. No two text lines within the edited text have the same line number. The text lines are stored in ascending sequence of their line numbers regardless of the order they were added to the edited text. The line numbers must not necessarily be consecutive. Notice that according to the productions given above, the first digit of a line number is

always the first character of the text line, no separators or other characters precede it.

If the line number consists of n digits, the separator contains $7-n$ blanks. Hence, list directive statements and PISA program text always start at the 8-th character of the text line.

The maximum length of a text line is implementation dependent but is never shorter than 80 characters.

C-4.2.2. List directive statements

If the first charcter following the separator is an exclamation mark (!), the text line contains a list directive statement. List directive statements may be inserted at any place within the edited text. They are not part of the PISA program text but serve to control the printout of a PISA source listing produced by the "list" command (C-4.5.).

There are two list directive statements defined for every PISA implementation: the space statement and the title statement. Additional list directive statements may be introduced by a PISA implementation. Invalid and unsupported list directive statements are ignored during a program listing printout without issuing a diagnostic message.

```
list_dir_stmt ::= (space_stmt | title_stmt) "•"
space_stmt   ::= "!" digit [digit]
title_stmt    ::= "!" title_text
title_text    ::= {character}
```

A space statement specifies that the indicated number of empty lines should appear at its place in the program listing. If there are fewer lines left on the actual page than specified in the space statement, a skip to the next page results. A space statement "!" + 99" always causes a skip to the next page.

A title statement specifies that the text given should appear on top of every page succeeding the title statement. The title text includes all characters following the colon up to but not including the end-of-line character. The title statement "!" + " results in a blank title. As long as no title statement is detected during editing of the PISA source listing, the title is blank.

C-4.2.3. PISA program text

If a text line does not have a "!" in its 8-th character, the line is assumed to contain PISA program text. The PISA program text includes all characters from the 8-th character up to but not including the end-of-line character. The syntax of the PISA program text held in the edited text is defined by the syntax of the programming language PISA (cf. Section B).

Examples:

```

70      const a = array (1..3) of string(5) •
80              (:'bolts','nuts','nails':) •
3507    print('ab c •
3508      def') •
730562 x := 0 •

```

The "print" procedure statement on the lines 3507 and 3508 contains an invalid literal string since it crosses the boundaries of a text line.

C-4.2.4. Ranges of text lines

Several of the commands available in edit mode require the specification of the range of text lines on which they are to be executed. These commands contain the nonterminal "range" in their syntax definition:

```

range      ::= single_line | sequence | all_lines
single_line ::= line_number
sequence   ::= from_line "." to_line
all_lines  ::= "*"
from_line  ::= line_number
to_line    ::= line_number

```

If the range specifies a single line, only that line is affected by the command containing the range specification. If a sequence of lines is specified, the first line (from_line) number must be less than the last line (to_line) number defining the range. The command is applied to all lines within the range in ascending order of line numbers. The range specification "*" denotes the sequence of all text lines within the edited text.

C-4.3. Modifying the edited text

The edited text can be modified by adding new lines to it, by deleting or changing existing lines, and by renumbering the edited text. New lines can be added to the text either by entering the lines at the terminal or by copying them from an existing S-library member into the text. The latter is requested by the "include" command.

Neither the syntax of the list directive statements nor the syntax of the PISA program text affected by a modification is checked when the modification is made. As explained in C-4.2.2., the list directive statements are never checked for syntactic correctness. The reason for not immediately checking the syntax of PISA program text is not so obvious and, therefore, is explained in more detail here.

Assume, for the moment, that we introduce immediate syntax checking of the PISA program text modified, and let us look at the implications of this policy.

As long as the PISA program text is entered sequentially in ascending order of line numbers, the implementation of immediate syntax checking is easy and straightforward. The parser is driven by the lines appended to the end of the existing program text and operates as usual. The main disadvantage of this method is that the parser occupies resources during all the time needed for entering a program. Another consequence is that the programmer typing the PISA text at a scope terminal is forced to periodically look at the screen in order to catch an eventual error message before the message is shifted out of the screen. Experience shows that most programmers do not like to divert their attention from the coding sheet and the keyboard; they prefer to enter a certain amount of program text and have it checked as a whole.

Let us now consider modifications to an existing and syntax-checked PISA program text. There are some modifications that involve no or no extensive syntax rechecking. Among these, we find insertion and deletion of simple statements, changes to constant values that do not alter their type, modifications of condition and action entries, etc. Modifications to other constructs, however, cause rechecking of larger parts of the program text: any change to a declaration part of a block, for example, generally results in a syntax recheck of the whole declaration part and the whole statement part of the block affected, which includes all nested blocks as well. Similar actions are caused by a modification to a label declaration.

Probably the most embarrassing case is the insertion or deletion of a construct that opens a block (if, begin, loop, etc.). Such a modification, in general, changes the block structure of all surrounding blocks until a missing or unmatched "end" is detected. This program restructuring makes necessary the rechecking of all affected blocks. In almost any case, the restructuring is only temporary, namely only as long as the corresponding "end" construct is not inserted or deleted, resp. Then, of course, the program's block structure is altered again and the altered part has to be rechecked anew. The first restructuring may easily result in a bulk of error messages made obsolete by the insertion or deletion of the "end" construct.

This enumeration of specific kinds of modifications to an existing PISA program text is not exhaustive at all. But it shows quite convincingly that the advantage of immediate syntax checks on PISA program text does not pay off the increased complexity of the parser and the augmented occupation of hardware resources.

In the previous paragraphs, we looked at the implications of immediate syntax checks. All considerations and statements made therein hold in a very similar way for immediate compilation as well. We can conclude that immediate compilation of PISA program text entered or changed in edit mode would lead to at least the same overhead compared to the well known method of a "deferred" compilation. A PISA program is compiled by the "run", "compile", "translate", and "check" commands introduced later in this chapter.

C-4.3.1. Adding a single text line

A single program line is added to the edited text by entering the line on the terminal.

```
add_one_line ::= line_number single_sep
               (list_dir_stat | pisa_prog_text)
single_sep   ::= " " | tabulation-character
```

Before the line entered is added to the edited text, it is adapted to conform to the syntax rules given in C-4.2.1.: all leading zeroes are erased from the line number and the single separator is replaced by n blanks, where n is equal to 7 minus the number of digits in the line number.

The single separator can be a single blank or a single tabulation. It is not part of the list directive statement or of the PISA program text, but all succeeding blanks and tabulations are.

Examples: (□ represents a tabulation character)

```

line entered:      37 goto abend•
line added:        37      goto abend

line entered:      0085□   x := 5•
line added:        85      x := 5

```

Error messages:

E100 line already exists

The line number entered already exists within the edited text. The line is not added to the text.

E101 line truncated

The line has been truncated to the maximum length defined by the PISA implementation.

E102 invalid line number

The line number contains characters other than the decimal digits, is longer than six digits, or is zero.

E103 maximum number of text lines exhausted

The edited text has reached its maximum length and cannot be expanded.

C-4.3.2. Adding a sequence of text lines

A sequence of text lines can be entered by adding them single line by single line as described in C-4.3.1. The add command relieves the programmer of entering line numbers. They are generated automatically.

```

add_cmd      ::= "a " first_line " " increment "•"
first_line   ::= line_number | "+"
increment    ::= digit {digit}

```

After a valid add command has been entered, the PISA system prompts the user to enter the next text line by printing out the

line number (without leading zeroes) and moving the cursor (or printing carriage) to the 8-th character of the text line. No standard prompter (underscore/backspace) is printed after the line number and the tabulation to column 8. The user may then enter the rest of the line belonging to the line number. After entering an end-of-line, the line will be added to the edited text exactly as described in C-4.3.1. (i.e., the same error messages may result, except *E102*).

The first line number generated is the one given by "first_line", where a "+" as first line specifies the highest existing line number in the edited text increased by the "increment". The "increment" must be greater than zero and is the difference between two successive line numbers generated.

All text entered following the automatically generated line number is stored as list directive statement or PISA program text. The only exception is the end command which stops the line number generation: if the text entered by the user consists of an "e" or "E" followed by an end-of-line, it is interpreted as end command. The line containing the end command is not added to the edited text.

Example:

```

add command:      a 170 10•
prenumbered lines: 170  if a = 0 then•
                   180      print('no values')•
                   190      end if•
end command:      200  e•
prompter:         -

```

Error message:

```

*E110* invalid add command
      Any syntax rule given for the add command has been
      violated.

```


C-4.3.3. Including text lines

A text that has been previously stored as a member into an S-library can be added to the edited text by the include command.

```
include_cmd    ::= "i " first_line " " increment " " member "."
first_line     ::= line_number | "+"
increment      ::= digit {digit}
```

The member specified is searched first in the user S-library, then in the project S-library, and finally in the system S-library. After it has been found, its text is retrieved line by line. The member must have been created by the "save" command (C-4.6.).

Before a text line is added to the edited text, the line number of the text line read from the member is changed according to the include command: The first line read gets the line number specified by "first_line", where a "+" as first line causes the first line number to be set to the highest line number in the edited text increased by the "increment". The increment must be greater than zero and is the difference between the line numbers of two successive lines added.

Example:

```
member c5:          50      print('asleep')
                   65      sleep(n)
                   80      print('awake')

highest line in
edited text:       970

include command:   i + 20 c5.
lines added:       990      print('asleep')
                   1010     sleep(n)
                   1030     print('awake')
```

Error messages:

```
*E120* invalid include command
      Any syntax rule given for the include command has
      been violated.
```

```
*E121* member not found
      The member specified was neither found on the user
      S-library, nor on the project S-library, nor on the
      system S-library.
```

- *E122* S-library read inhibited -- try later
One of the S-libraries cannot be accessed now.
- *E123* illegal member name
The member name is syntactically wrong.
- *E124* line number conflict -- include terminated
The line number of the next text line to be added either already exists in the edited text, or there is an existing line between the previous line added and the one to be added.
- *E125* invalid line number format -- include terminated
This error occurs only if the member has not been created by the "save" command. It indicates that the syntax of a text line has been violated (cf. C-4.2.1.).
- *E126* maximum number of lines exhausted -- include terminated
The edited text has reached its maximum length and cannot be expanded.

The include operation stops immediately after an error has been detected. At end of the include operation, whether it terminates normally or with an error, one of the following two messages is printed on the terminal:

```
no lines included
last line included is <n>
```

These messages succeed an eventual error message.

<n> is replaced by the line number of the last line that has been added successfully. It is the line number within the edited text, not the one within the member included.

C-4.3.4. Deleting text lines

The delete command serves to delete a range of text lines.

```
delete_cmd      ::= "d " range "•"
```

The range specification has been defined in C-4.2.4.

If the delete command will delete more than 10 text lines, the user must confirm the delete request by answering the message

really delete more than 10 lines?

with any string starting with a "y" or "Y". Otherwise, no lines are deleted.

Example:

```
old text:          130      a := 0
                   140      b := 5
                   150      n := 'first'
                   160      init(x)
delete command:    d 140..160*
new text:          130      a := 0
```

Error messages:

```
*E130* invalid delete command
      Any syntax rule given for the delete command has
      been violated.
```

```
*E131* line does not exist
      The range specification indicated a single line to
      delete and this line does not exist within the
      edited text.
```

C-4.3.5. Changing text lines

The change command allows to change the text of a range of text lines.

```
change_cmd      ::= (change_all | change_n-th) range
                  delim old_text delim new_text "*"
change_all      ::= "c "
change_n-th     ::= "c" digit [digit] " "
delim           ::= character
old_text        ::= character {character}
new_text        ::= {character}
```

The range specification has been defined in C-4.2.4.

The character separating the range specification from the old text is called delimiter. The delimiter can be any character except a blank and must not be contained in the old text. The first delimiter found to the right of the one preceding the old text terminates the old text. All characters following this second delimiter up to but not including the end-of-line character constitute the new text. Hence, if the new text contains the delimiter, it is part of the new text.

The change all version of the change command searches the old text in the lines within the range specified and replaces all occurrences of the old text by the new text. No part of the new text is searched for a match with the old text after the new text has been inserted in place of the old one. This is illustrated by:

```
old line:      750    a := a + 1
change command: c 750/a/ca*
new line:      750    ca := ca + 1
```

The change n-th occurrence version searches the old text in the lines within the range and replaces the n-th occurrence of the old text within every text line by the new text. n is specified by a one or two digit number immediately following the "c" indicating a change command (n>0).

```
old lines:      300    a := a + b
                 310    aa := a + b
                 320    a := x + a
change command: c2 300..320,a,ac*
new lines:      300    a := ac + b
                 310    aac := a + b
                 320    a := x + ac
```

The change operation does only inspect the part of the text lines starting at column 8. Therefore, the line number and the separator can never be changed.

Each line modified as a result of a change command is printed out on the terminal in its changed form.

Error messages:

```
*E140* invalid change command
      Any syntax rule given for the change command has
      been violated.
```

***E141* line truncated**

This error message follows the printout of the line that has been truncated to the maximum line length defined by the implementation.

C-4.3.6. Renumbering the edited text

The renumber command allows to renumber the edited text.

```
renumber_cmd  ::= "r " first_num " " increment "•"
first_num     ::= line_number
increment     ::= digit {digit}
```

The edited text is renumbered by assigning the line number n to the i -th line in the edited text, where $n = \text{first_num} + ((i-1) * \text{increment})$. Before the renumbering starts, PISA makes sure that the last line number assigned is not greater than 999999 (cf. error message E151). The increment specified must be greater than zero.

Example:

```
old text:          20      main procedure ha
                   1570      print ('that's it')
                   13862     end procedure ha

renumber command:  r 100 200•
new text:          100      main procedure ha
                   300      print ('that's it')
                   500      end procedure ha
```

Error messages:

***E150* invalid renumber command**

Any syntax rule given for the renumber command has been violated.

***E151* line numbers would exceed 999999**

If the text were renumbered as requested, there would be at least one line number greater than 999999.

C-4.4. Inspecting the edited text

The edited text may be inspected by requesting a printout of a specified portion of its text lines on the terminal. The print command is used to request a printout of a range of lines; the verify command serves to print all lines in a range having at least one occurrence of a certain substring.

C-4.4.1. Printing a range of text lines

The print command is used to request a printout of the text lines within the range specified.

```
print_cmd      ::= "p " range "•"
```

The range specification has been defined in C-4.2.4.

```
Examples:      p 305•
                p 150..220•
                p *•
```

Error message:

```
*E200* invalid print command
      Any syntax rule given for the print command has
      been violated.
```

C-4.4.2. Printing selected text lines

The verify command serves to request a selective printout of those lines within a certain range having at least one occurrence of a specified substring.

```
verify_cmd     ::= "v " range delim substring "•"
delim          ::= character
substring      ::= character {character}
```

The range specification has been defined in C-4.2.4.

The character separating the range specification from the substring is called delimiter. The delimiter can be any

character except a blank and may be contained in the substring. All characters following the delimiter up to but not including the end-of-line constitute the substring.

Every line within the range specified is inspected for an occurrence of the substring. If it is found anywhere between the 8-th character of the text line and the end-of-line, then the text line is printed out, else it is ignored.

Example:

```

edited text:      640      var h: string(5)
                  650      const alpha = 15.3982
                  660      type help = (yes,no,perhaps)
verify command:   v 640..660/ h•
printed lines:    640      var h: string(5)
                  660      type help = (yes,no,perhaps)

```

Error message:

```

*E210* invalid verify command
      Any syntax rule given for the verify command has
      been violated.

```

C-4.5. Producing a program listing

The print command, as has been described above, may be used to request a printout of the edited text on the terminal. The text is printed as it is stored in the edited text. The list command is used to request a formatted source listing of the PISA program held in the edited text.

```

list_cmd      ::= "list " routing [ " " options] "•"
routing       ::= "term" | device_address
options       ::= implementation_dependent_options

```

The routing information specifies where the listing has to be printed out. If the routing indicates "term" (either in small or in capital letters), the listing appears on the terminal where the list command has been entered. Otherwise, the routing is an implementation dependent address designating any output device connected to the computer and available to the PISA session.

The options are implementation dependent and may be used to control certain features of the list producing program.

Examples: list term•
 list remote6•
 list LP3 noformat,pagesize=48•

The design of the program listing's format is left to the PISA implementor. The only requirements are (1) that the listing includes all text lines within the edited text in ascending order of their line numbers, and (2) that the two list directive statements introduced in C-4.2.2. are correctly processed. It is expected, however, that the program listing is formatted as follows (perhaps controllable by options):

- The lines are indented according to the block structure of the program.
- The range of blocks is shown by a nesting level indicator or by vertical lines inserted into the program text.
- The condition and action entries are shifted to start at a unique column of the listing (e.g. column 80).
- A cross-reference of identifiers follows the listing. For each identifier, the line number of its declaration is given together with the line numbers of its denotations. If an identifier is used as a target variable in an assignment, as actual variable parameter, or as actual area parameter, the line number containing such a denotation is marked as a "target" reference.
- All decision tables within the program are analysed for completeness, redundancies, ambiguities, implications, and contradictions. Related diagnostics are included in the source listing.

Error messages:

- *E220* invalid list command
 Any syntax rule given for the list command has been violated.
- *E221* invalid routing
 The routing specified an invalid output address or designated an output device not available to the PISA session.

E222 device occupied -- try later
 The routing specified a valid output device, but this device is currently in use and no spooling is performed for it.

E223 invalid option(s)
 One or more of the options specified are invalid.

C-4.6. Saving the edited text

The save command serves to store the edited text as a member into the user S-library. The saving does not change the contents of the edited text.

save_cmd ::= "save " member "•"

If the member already exists on the user S-library, the user must confirm the save request by answering the message

member already exists -- really overwrite it?

with any string starting with a "y" or "Y". Otherwise, the text is not saved.

Error messages:

E230 invalid save command
 Any syntax rule given for the save command has been violated.

E231 S-library write inhibited -- try later
 The user S-library cannot be modified now. The save request is not executed.

E232 illegal member name
 The member name specified is syntactically wrong.

C-4.7. Compiling the edited text

The compile command is used to produce C-code from the PISA program or external routine in the edited text. The C-code (compiled code) is an intermediate code executed by the C-code interpreter (see C-5.).

```
compile_cmd ::= "compile" [ " s" ] "*" "
```

The program errors detected during compilation are reported to the user by writing so called compile-time error messages on the terminal. If the compile command does not include the "s" option ("s" for short error messages), a compile-time error message consists of the following four lines:

The first line is the text line containing the error.

The second line marks the erroneous symbol or construct by underlining its first character with an asterisk "*". In the cases where the error message refers to a missing construct preceding a certain symbol, the second line contains a "<" character below the first character of the symbol following the missing construct.

The third line is the error message itself. It has the format

```
*P<n>* <error text>
```

where <n> is replaced by a three digit error number and <error text> by a short error description. All PISA compile-time errors are described in Appendix C-IV together with their error number.

The fourth line is a blank line, i.e. it consists of an end-of-line character only.

If the compile command is entered as "compile s*", this indicates short compile-time error messages. A short compile-time error message consists of one line only: it is the line number of the erroneous line followed by the error message (the third line above) starting at column 8.

Examples:

```
370    balance := old_balance + account
      *
*P037* undeclared identifier

5200          x: array (1..5 of index
              <
*P020* parenthesis missing

3560    *P127* procedure identifier missing
```

If a compile-time error has been detected, the C-code generated is not linked and not written to the user C-library.

If the compilation did not reveal any error, the message

no errors detected

is written on the terminal before the external procedures and function declarations are resolved by linking the corresponding external routines to the C-code generated. They are searched as members first in the user C-library, then in the project C-library, and finally in the system C-library. The member name is equal to the procedure or function name, resp.

After an errorless compilation and successful linking of all external procedures and functions, the resulting C-code is stored as a member into the user C-library. The name of the member is the declared name of the main procedure, external procedure, or external function, i.e. the name of the outermost PISA block in the edited text. If the user C-library already contains a member with this name, the user must answer the message

member already exists -- really overwrite it?

with any string starting with a "y" or "Y". Otherwise, the C-code is not stored.

Error messages:

E240 invalid compile command
Any of the syntax rules given for the compile command has been violated.

E241 member <m> not found -- unable to link
where <m> is replaced by the name of the member not found. This member is neither present on the user C-library, nor on the project C-library, nor on the system C-library.

E242 C-library write inhibited -- try later
The user C-library cannot be modified now. The compile request is abnormally terminated.

E243 C-library read inhibited -- try later
Any of the C-libraries cannot be accessed during linking. The compile request is abnormally terminated.

C-4.8. Translating the edited text

The `translate` command serves to translate the PISA program or external routine held in the edited text into T-code (cf. C-6.).

```
translate_cmd ::= "translate" [" s"] "•"
```

The errors detected during translation are exactly the same as in a compilation (cf. C-4.7.), and the format of the compile-time error messages is the same as described in C-4.7. also. The "s" option indicates short compile-time error messages (cf. C-4.7.).

If the translation did not reveal any error, the message

```
no errors detected
```

is printed on the terminal and the external procedure and function declarations are resolved by linking the corresponding external routines to the T-code. They are searched as members first in the user T-library, then in the project T-library, and finally in the system T-library. The member name is equal to the procedure or function name, resp.

After an errorless translation and successful linking of all external procedures and functions, the resulting T-code is stored as a member into the user T-library. The name of the member is the declared name of the main procedure, external procedure, or external function, i.e. the name of the outermost PISA block in the edited text. If the user T-library already contains a member with this name, the user must answer the message

```
member already exists -- really overwrite it?
```

with any string starting with a "y" or "Y". Otherwise, the T-code generated is not stored.

Error messages:

```
*E250* invalid translate command
```

```
Any of the syntax rules given for the translate
command has been violated.
```

```
*E251* member <m> not found -- unable to link
```

```
where <m> is replaced by the name of the member not
found. This member is neither present on the user
T-library, nor on the project T-library, nor on the
system T-library.
```

E252 T-library write inhibited -- try later
 The user T-library cannot be modified now. The
 translate request is abnormally terminated.

E253 T-library read inhibited -- try later
 Any of the T-libraries cannot be accessed during
 linking. The translate request is abnormally
 terminated.

C-4.9. Checking the edited text

The check command serves to check the PISA program or external
 routine held in the edited text for syntactic correctness.

```
check_cmd      ::= "check" [" s"] "s"
```

The action caused by the check command is exactly the same as
 defined for the compile command (cf. C-4.7.), except that the
 C-code produced is not stored into the user C-library. The "s"
 option indicates short compile-time error messages (cf. C-4.7.).

Error messages:

E241 member <m> not found -- unable to link
 where <m> is replaced by the name of the member not
 found. This member is neither present on the user
 C-library, nor on the project C-library, nor on the
 system C-library.

E243 C-library read inhibited -- try later
 Any of the C-libraries cannot be accessed during
 linking. The check request is abnormally
 terminated.

E260 invalid check command
 Any of the syntax rules given for the check command
 has been violated.

C-4.10. Running the program

The run command is used to run the PISA program held in the
 edited text.

```
run_cmd      ::= "run" [ " s" ] [ " :=" comm_string ] "*"
comm_string  ::= {character}
```

The program text in the edited text must be a PISA main procedure. The run command causes compilation and linking of the PISA program exactly as described in C-4.7. The "s" option indicates short compile-time error messages (cf. C-4.7.). The text given as "comm_string" is assigned to the communication string (cf. SSR 1004 and SSR 4000 in Appendix B-II) before the program execution starts. If this option is not included in the run command, the communication string is not changed by the run command.

In opposition to the compile command, the C-code produced is not written into the user C-library but stored into the session's program area. If neither the compilation nor the linking revealed an error, the session is switched from edit mode to run mode (cf. C-5.).

After termination of the program, the mode is switched back from run mode to edit mode. The edited text is still available, i.e., it contains the same text lines as before the session was switched to run mode.

Error messages:

- *E241* member <m> not found -- unable to link
 where <m> is replaced by the name of the member not found. This member is neither present on the user C-library, nor on the project C-library, nor on the system C-library.
- *E243* C-library read inhibited -- try later
 Any of the C-libraries cannot be accessed during linking. The run request is abnormally terminated.
- *E270* invalid run command
 Any of the syntax rules given for the run command has been violated.
- *E271* no main procedure
 The program text held in the edited text is not a PISA main procedure (cf. B-9.1.). The run command is not executed.

C-5. THE RUN MODE

=====

The run mode executes the C-code in the session's program area. The C-code can be a PISA program loaded from a C-library in system mode, a PISA program compiled and linked in edit mode, or a PISA statement entered for immediate execution and compiled in halt mode.

C-5.1. The basic structure of the run mode

When the run mode is entered, it starts executing the C-code in the session's program area without writing any text to the terminal. Every time the run mode is entered, the user interrupt facility is enabled.

Execution of the C-code continues until one of the following events occurs:

- (1) If the execution reaches a C-code instruction marking the normal end of a PISA program, the session is switched back to the mode that placed the C-code into the program area (either system mode or edit mode).
- (2) If the execution reaches a C-code instruction marking the normal end of a PISA immediate statement (cf. C-7.3.), the session is switched back to halt mode.
- (3) If a run-time error is detected, a run-time error message (C-5.3.) is written to the terminal and the session switched to halt mode.
- (4) If the predefined procedure stop (B-10.6.) is invoked, the run-time error message R001 (C-5.3.) is written to the terminal and the session switched to halt mode.
- (5) An enabled user interrupt during run mode causes:
 - a) discontinuation of the execution after a user interruption location has been reached (cf. C-5.4.3.),
 - b) the error message R002 (C-5.3.) to be printed on the terminal, and

c) the session to be switched to halt mode.

When the user interrupt facility is disabled, all user interrupts are ignored.

As long as none of the events described above occurs, the terminal is completely under the control of the running PISA program. Whenever the run mode is left, the echo switch is set to "on" and the user interrupt facility is enabled (cf. B-11.1.).

C-5.2. The C-code

The C-code, or compiled code, is an intermediate code produced by compilation of PISA program text and executed by the C-code interpreter. The format of the C-code is not defined; the PISA implementor may adapt it to the underlying hardware and system software.

The C-code interpreter enforces all PISA language rules that have not been verified during compilation of the PISA program into C-code. It does in no way assume a correct program and checks everything for correctness that could possibly be incorrect. Any violation of a language rule is caught immediately when it occurs and is reported to the user by a run-time error message. All run-time error messages are related to the definition of the programming system PISA and designate the location in the source program text where the error has been detected (see also E-2.1.).

The C-code and the C-code interpreter are not designed for fast program execution and for small main storage requirements but for comprehensive error detection and testing support.

If any abnormal system error occurs during run mode, the PISA run-time system writes the error message

***R999* PISA system error**

on the terminal. This message should be followed by one or more lines of information that help trace the error. After the messages have been produced, the implementation is free to either reestablish the session in halt, edit, or system mode, or to cancel the session and switch it to the idle state. The preferred version is to treat such an error in the same way as a usual run-time error.

C-5.3. Run-time error messages

The format of the run-time error messages is defined by:

```
run_err_msg    ::= error_# " " error_text [err_location]
error_#        ::= "*" digit digit digit "*"
error_text     ::= character {character}
err_location   ::= " at line " line_# " in " routine_id
line_#         ::= digit {digit}
routine_id     ::= character {character}
```

The error location is present in all run-time error messages except in the error R999 and in run-time errors detected within the C-code generated by an immediate statement (cf. C-7.3.).

Examples:

```
*R001* invalid code at line 326 in invlist
*R002* user interrupt at line 8200 in prog102
*R185* division by zero
```

The error numbers are classified into three groups:

```
*R001* - *R499*   Run-time errors defined for every PISA
                  implementation. These errors are defined
                  in Appendix C-V.

*R500* - *R998*   Implementation dependent run-time errors.
                  They are defined in the implementation
                  handbook.

*R999*           Abnormal system error during run mode (see
                  C-5.2.).
```

The error texts for the implementation dependent error messages are given in Appendix C-V. Notice that the error text for the error R001 (caused by the "stop" procedure statement) is the string passed as actual parameter to the predefined procedure "stop" (cf. B-10.6.).

The line number gives the number of the text line where the statement that caused the run-time error starts. If the error was a user interrupt (R002), it is the line number of the interruption location (cf. C-5.4.3.).

The routine identifier is the name of the program or external routine (P-9.) wherein the run-time error occurred.

C-5.4. Interruption locations

Events (3), (4), and (5) as described in C-5.1. cause an abnormal interruption of the PISA program in execution. In these cases, an interruption location is set prior to switching the session to halt mode. The interruption location is defined below for every kind of abnormal interruption.

C-5.4.1. Run-time errors

PISA run-time errors can be classified into four categories according to the action causing them:

- (1) Expression evaluation errors
- (2) Constant and variable denotation errors
- (3) Procedure and function seizing errors
- (4) Errors in predefined procedures

C-5.4.1.1. Expression evaluation errors

Expression evaluation errors may occur within a statement or in a binding declaration and result from:

- a) Illegal values of operands (uninitialized variables, noninteger exponents for negative bases, division by zero, etc).
- b) Type adaptation or conversion of an operand, of an intermediate value, or of the expression's final value failed (cf. B-2.5.). This includes illegal index values for indexed constants and variables as well as invalid actual value parameters.
- c) Inability to perform a numeric operation on the implementation (overflows, underflows, etc.).

d) Error during evaluation of a predefined function.

Only a subset of all PISA statements can cause an expression evaluation error: assignment, procedure, if, case, decision table, and repetitive statements. Errors within component statements of a structured statement are not considered to be errors of the structured statement. Notice that the definition of the repetitive (loop) statement (B-6.3.3.) implies that an eventual run-time error occurs during evaluation of the initial and final value only, never during incrementing or decrementing the control variable. If an expression evaluation error is detected during execution of one of the statements mentioned above, the interruption location is set immediately preceding the failing statement. This is done independent on how much of the statement has already been evaluated; the user is responsible to take care of all side-effects. The implementation makes sure that no illegal state of any internal control structure is left by the run-time error. If the failing statement is a decision table statement, the interruption location is set preceding the statement regardless of which condition caused the error.

A binding declaration may trigger an expression evaluation error only if binding is to an indexed variable. If the erroneous binding declaration is contained in the declaration part of a block constituted by a structured statement, the interruption location is set preceding the statement that constitutes the block. In the case of a decision table or loop statement, an arbitrary number of block activations (B-5.1.) may have preceded the activation leading to the error. If the failing binding declaration is part of a procedure block, the procedure is released (B-7.1.) and the interruption location set preceding the invoking procedure statement. If the binding failed in a function block, the function is released (B-8.1.) and the interruption location set at the same place as for an evaluation error of the expression containing the function designator (see above).

C-5.4.1.2. Constant and variable denotation errors

Constant and variable denotation errors may occur because of:

- a) An illegal substring constant denotation (cf. B-3.2.).
- b) An illegal substring variable denotation (cf. B-3.4.).
- c) A denotation of a nonexistent referenced variable (cf. B-3.4.).

Notice that invalid index expressions in denotations of indexed constants and indexed variables are treated as expression evaluation errors (C-5.4.1.1.).

The denotation error may appear within a statement (in an expression, in an actual variable parameter, in an actual area parameter, and on the left hand side of an assignment) or within a binding declaration. The interruption location for these two cases is defined to be at the same place as given for expression evaluation errors (see C-5.4.1.1.).

C-5.4.1.3. Procedure and function seizing errors

A seizing error is raised when a procedure or function cannot be seized (B-7.1., B-8.1.). This can be due to a lack of storage, to an overflow of some internal table, or to recursive coroutine invocation.

If seizing of a procedure fails, the interruption location is set preceding the invoking procedure statement. If a function cannot be seized, the interruption location is set at the same place as for an evaluation error of the expression containing the invoking function designator (cf. C-5.4.1.1.).

C-5.4.1.4. Errors in predefined procedures

If a run-time error occurs within a predefined procedure (including the sys-procedure), the interruption location is set preceding the invoking procedure statement. Remember that errors in predefined functions are treated as expression evaluation errors.

Since non-PISA procedures are invoked via a system service request, run-time errors within them are treated in the same way as run-time errors within a system service request itself.

C-5.4.2. Predefined procedure "stop"

After abnormal interruption of a PISA program caused by an invocation of the predefined procedure "stop" (B-10.6.), the interruption location is set immediately following the invoking procedure statement.

C-5.4.3. User interrupt

A user interrupt is not a "hard" interrupt, it merely sets a flag that can be inspected by the C-code interpreter. After the flag has been set, the C-code interpreter continues execution until it reaches a point within the PISA program where a PISA statement may be legally inserted. The interruption location is set at this place. If, after setting the interrupt flag but before reaching the interruption location, another abnormal interruption occurs, the flag is cleared and the other interruption treated as defined.

If a user interrupt is entered during the time the program waits for input from a real or pseudo terminal (cf. B-11.1. and SSR 2004 in Appendix B-II), the interrupt is treated as an error in a predefined procedure and the interruption location set as described in C-5.4.1.4.

C-5.4.4. Block termination

If the interruption location is set succeeding the last statement of a block, the block is not yet terminated (B-5.1.). It will be terminated as soon as the interruption location is cleared. As explained under "halt mode" (C-7.), this is caused either by the continue command or by immediate execution of a goto or exit statement.

C-6. THE EXECUTE MODE

=====

The execute mode executes the T-code in the session's program area. The T-code is a program loaded from a T-library in system mode. The execute mode is only entered from the system mode and always passes control back to the system mode.

C-6.1. The basic structure of the execute mode

When the execute mode is entered, it starts execution of the T-code in the session's program area without writing any text to the terminal. Every time the execute mode is entered, the user interrupt facility is enabled.

Execution of the T-code continues until one of the following events occurs:

- (1) If the execution reaches a T-code instruction marking the normal end of a PISA program, the session is switched back to system mode.
- (2) If an execution error is detected, an execution error message (C-6.3.) is written to the terminal and the session switched to system mode.
- (3) If the predefined procedure stop is invoked, the execution error message X001 (C-6.3.) is written to the terminal and the session switched to system mode.
- (4) An enabled user interrupt during execute mode causes:
 - a) discontinuation of the execution as soon as possible,
 - b) the error message X002 (C-6.3.) to be printed on the terminal, and
 - c) the session to be switched to system mode.

When the user interrupt facility is disabled, all user interrupts are ignored.

As long as none of the events described above occurs, the terminal is completely under the control of the PISA program in execution. Whenever the execute mode is left, the echo switch is set to "on" and the user interrupt facility is enabled (cf. B-11.1.).

C-6.2. The T-code

The T-code (translated code) of a PISA program is produced by the "translate" command (C-4.8.). The format of the T-code is undefined; it can be an intermediate code or machine code. If it is intermediate code, it is executed by some "soft" interpreter; if it is machine code, its interpreter is the hardware itself.

The T-code interpreter assumes a correct PISA program and, therefore, does not enforce all PISA language rules. The only checks performed are those required to ensure safe operation of the entire system and those done outside the PISA software (by the hardware and by the operating system). Hence, the program is not protected from its own errors as long as they do not affect other system components. If an error is detected during execution of the T-code, an execution error message is written on the terminal, the program immediately terminated, and the session switched back to system mode. The execution error messages must designate the line number within the original source program text wherein the error was detected. The message must not necessarily be related to the definition of the programming system PISA (see also E-2.2.); the implementation handbook may describe the reasons that can lead to specific execution errors.

The T-code and its interpreter (hardware or software) are trimmed for execution speed and explicitly not for comprehensive error detection and testing support.

If any abnormal system error occurs during execute mode, the PISA run-time system writes the error message

X999 PISA system error

on the terminal. This message should be followed by one or more lines of information that help trace the error. After the messages have been produced, the implementation is free to either reestablish the session in system mode, or to cancel the session and switch it to the idle state. The preferred version is to treat such an error in the same way as a usual execution error.

C-6.3. Execution error messages

The format of the execution error messages is defined by:

```

exec_err_msg  ::= error_# [ " " error_text ] [err_location]
error_#       ::= "**X" digit digit digit "*"
error_text    ::= character {character}
err_location  ::= " at line " line_# " in " routine_id
line_#        ::= digit {digit}
routine_id    ::= character {character}

```

The error location is present in all execution error messages except in the error X999.

Examples:

```

*X001* invalid code at line 326 in prog3
*X002* user interrupt at line 8200 in sales_report
*X185* at line 650 in file036
*X508* invalid database structure at line 203 in join

```

The error numbers are classified into three groups:

```

*X001* - *X002*   Execution errors defined for every PISA
                   implementation. Error X001 is caused by
                   event (3) as described in C-6.1., error
                   X002 by event (4).

*X003* - *X998*   Implementation dependent execution errors.
                   Except for the errors X001, X002, and X999,
                   all execution errors are implementation
                   dependent.

*X999*            Abnormal system error during execute mode
                   (see C-6.2.).

```

The error texts are only defined for three execution errors:

```

*X001* <stop text>
*X002* user interrupt
*X999* PISA system error

```

The <stop text> is the string passed as actual parameter to the predefined procedure "stop" (cf. B-10.6.). For all other execution errors, the implementation is free to print out error

texts or to give an error number only. If no text is printed, the errors must be explained in the implementation handbook.

The line number gives the number of the text line where the statement that caused the execution error starts. If the error was a user interrupt (X002), it is the line number where the last executed statement started.

The routine identifier is the name of the program or external routine (B-9.) wherein the execution error occurred.

C-7. THE HALT MODE

=====

The halt mode is entered from run mode after the running PISA program is abnormally interrupted. As described in C-5.1., abnormal interruption can be caused by a run-time error, by invoking the predefined procedure "stop" (cf. B-10.6.), or by an enabled user interrupt. In halt mode, the user may enter halt commands and PISA statements for immediate execution. The main purpose of the halt mode is to find the reason for a program malfunction with the actual state of the program retained at the point of the abnormal interruption. If desired, the execution of the program can be resumed from the interruption point after an arbitrary number of immediate statements have been executed. If the execution is not to be resumed, the session may be switched to edit mode or system mode.

Whenever the halt mode is active, an interruption location is associated with it. It is called the actual interruption location. The interruption location is "set" in run mode before the session is switched to halt mode; the interruption location is "cleared" by either a continue command or by execution of an immediate goto or exit statement transferring control out of the immediate statement.

This chapter defines the dialogue of the halt mode. It introduces the halt commands and gives the rules for immediate statements. The implementor of PISA is free to introduce additional halt commands. The dialogue as defined below, however, must be supported on every PISA implementation.

C-7.1. The basic structure of the halt mode

During halt mode, the terminal is controlled by the PISA run-time system. The basic structure of the halt mode is defined by the PISA program below.

```
main procedure halt_mode
```

```
const backspace =      8  (* ASCII BS *)
var line:          string(255) vary
```

```
interrupt(true)
```

```

*> next_command:

printnr('_',val(char,backspace))
input(line)

if small(line) = 'sys' then
    (* switch session to system mode, cf. C-7.2.1. *)
end if

if small(line) = 'edit' then
    (* switch session to edit mode, cf. C-7.2.2. *)
end if

if small(line) = 'cont' then
    (* continue execution of program by switching *)
    (* the session back to run mode, cf. C-7.2.3. *)
end if

if small(line) = 'loc' then
    (* print line number and routine name of active *)
    (* interruption location, cf. C-7.2.4. *)
    goto next_command
end if

(* line entered is considered as PISA text that is *)
(* to be executed immediately (cf. C-7.3.). Com- *)
(* pile the statements to C-code; if no errors de- *)
(* tected, switch the session to run mode, other- *)
(* wise continue below. *)

goto next_command

end procedure halt_mode

```

As this description of the halt mode's basic structure shows, the user interrupt facility is enabled each time the halt mode is entered. A user interrupt during halt mode

- (1) discontinues the current activity and does the necessary clean-up and
- (2) passes control to the first statement in the procedure "halt_mode".

If the immediate PISA statement to be entered happens to be textually identical with one of the halt commands, it must be

preceded by one or more PISA separators (cf. B-1.4.). This prevents the line from being treated as a halt command.

If any abnormal error occurs in halt mode, the PISA run-time system writes the error message

```
*H999* PISA system error
```

on the terminal. This message should be followed by one or more lines of information that help tracing the error. After the messages have been produced, the implementation is free to either reestablish the session in halt mode, edit mode, or system mode, or to cancel the session and switch it to the idle state. The preferred version is to treat such an error in the same way as a usual halt mode error.

C-7.2. Halt commands

The halt commands defined below are supported on every PISA implementation.

```
halt_cmd      ::= sys_cmd | edit_cmd | cont_cmd | loc_cmd
```

C-7.2.1. Switching the session to system mode

The sys command serves to switch the session from halt mode to system mode. This command is only accepted if the C-code in the session's program area has been loaded in system mode (cf. C-3.1.). Otherwise, the error H001 is printed on the terminal and control passed to the label "next_command" in the procedure "halt_mode" (C-7.1.).

```
sys_cmd       ::= "sys•"
```

Error message:

```
*H001* run mode has been activated from edit mode
      There is still a PISA program in the edited text
      (C-4.2.). The session can only be switched to edit
      mode or run mode.
```

C-7.2.2. Switching the session to edit mode

The edit command serves to switch the session from halt mode to edit mode. This command is only accepted if the C-code in the session's program area has been loaded in edit mode (cf. C-4.10.). Otherwise, the error H002 is printed on the terminal and control passed to the label "next_command" in the procedure "halt_mode" (C-7.1.).

```
edit_cmd      ::= "edit•"
```

Error message:

```
*H002* run mode has been activated from system mode
      There is no PISA program in the edited text
      (C-4.2.). The session can only be switched to
      system mode or run mode.
```

C-7.2.3. Continuing execution of interrupted program

The continue command is used to switch the session back to run mode. This causes the execution to be resumed at the actual interruption location (C-5.4.). This interruption location is cleared.

```
cont_cmd      ::= "cont•"
```

C-7.2.4. Requesting the interruption location

If the user has any doubts about where the actual interruption location is, he can request it with the location command.

```
loc_cmd       ::= "loc•"
```

The PISA run-time system responds with the message

```
at line <n> in <r>
```

<n> indicates the line number of the actual interruption location. It is the line number on which the statement succeeding the interruption location starts. If the interruption

location is set succeeding the last statement in a block, it is the line number where the construct that terminates the block starts. <r> is the name of the program or external routine (B-9.) wherein the actual interruption location is set.

C-7.3. Immediate PISA statements

The basic structure of the halt mode exhibits that if a line entered in halt mode is not a valid halt command, the line is assumed to be PISA program text to be executed immediately. The syntax of such a line is defined by:

```
imm_stmt      ::= {simple_stmt | struc_stmt} "•"
```

where the syntax of simple and structured statements is defined in Section B. Statements entered in halt mode are called immediate statements. The following syntactical restrictions and conventions apply for immediate statements:

- (1) The immediate statements may not be continued on another line. The end-of-line always terminates the text.
- (2) The declaration part of blocks constituted by immediate structured statements must be empty.
- (3) The immediate statements are virtually inserted at the actual interruption location. This location determines the denotable objects according to the scope rules defined for the programming language PISA.

After the immediate statements have been entered, they are compiled to C-code with the above conventions in mind. As soon as the compilation detects an error, a compile-time error message is written on the terminal, the rest of the line entered ignored, and control passed to the label "next_command" in the procedure "halt_mode" (C-7.1.).

The format of the compile-time messages produced in halt mode is essentially the same as defined under "compiling the edited text" in C-4.7.; the difference is that only the second and third line of the message are produced in halt mode. Example:

```

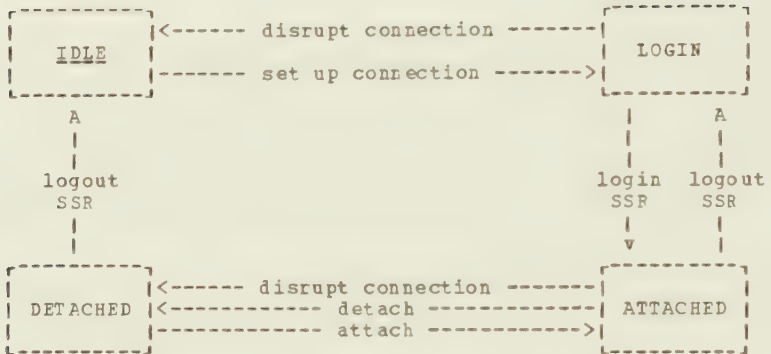
line entered:          print (balnace)
error message:         *
                        *P037* undeclared identifier
prompter:              -

```

If the immediate statements have been successfully compiled to C-code, the session is switched to run mode for their execution. After they have been executed, event (2) as described in C-5.1. occurs and the session is switched back to halt mode. There are, however, a few cases that require special treatment:

- If a goto statement is executed within the immediate statements, the actual interruption location is cleared and execution of the interrupted PISA program or external routine continues at the place of the designated label.
- If an exit statement within the immediate statements designates a block surrounding the actual interruption location, the interruption location is cleared and execution of the interrupted PISA program or external routine continues.
- The immediate statements may involve parts of the PISA program and external routines in the session's program area during their execution. This happens if a suspend statement is executed within them, or if they invoke a non-predefined procedure or function. Control flow and parameter passing between the immediate statement and the program or external routine is defined by the programming language PISA (Section B). If the execution triggered by immediate statements is abnormally interrupted (events (3), (4), and (5) in C-5.1.), a run-time error message is written on the terminal (cf. C-5.3.). If the resulting interruption location is set within the immediate statements, the session is switched to halt mode and the interruption location as well as the remaining immediate statements are disregarded. If the interruption location is within the program or within an external routine, this location becomes the actual interruption location before the session is switched to halt mode. The previous interruption location(s) will become actual as soon as control returns to the immediate statement that transferred control out of it.

Appendix C-I: Session state transition graph



IDLE is the initial and final state. Transitions in the horizontal direction indicate changes in the physical connection between the session and the terminal. Transitions in the vertical direction indicate changes from logged out to logged in state and vice versa.

Appendix C-II: Session mode transition graph



Appendix C-III: Sample session

All end-of-line characters entered are represented by the fat dot "." in the example below.

```

login
FISA account? 15,13.
password? .
system mode
edit.
edit mode
a 10 10.
10   main procedure eratosthenes.
20   var s: array (2..1000) of boolean.
30   var i,j: 2..2000.
40   loop init for i:=2 to 10000 s(i):=tru end loop init.
50   loop fill_sieve for i:=2 to 500.
60       if s(i) then.
70           j := i + i.
80           loop mark_multiples while j<=1000.
90               s(j) := false.
100              j := j + 0.
110          end loop mark_multiples.
120      end if.
130  end loop fill_sieve.
140  loop print_primes for i:=1 to 1000.
150      if s(i) then print(i) end if.
160  end loop print_primes.
170  end procedure eratosthenes.
180  e.
save erat.
run.
40   loop init for i:=2 to 10000 s(i):=tru end loop init
                                     *
*P037* undeclared identifier

90           s(j) := false
              <
*F012* ) missing

c 40/tru/true.
40   loop init for i:=2 to 10000 s(i):=true end loop init
c 90/j/j) .
90           s(j) := false
run.

```

```

nc errors detected
*R008* type adaptation failed at line 40 in eratosthenes
print(i)•
1001
exit init•
*R002* user interrupt at line 90 in eratosthenes
trace(1)•
cont•
90      100      90      100      90      100      90      100      90
100      90      100      90      100      90      100      90      100
90      100      90
*R002* user interrupt at line 100 in eratosthenes
print(i,' ',j)•
2 4
edit•
edit mode
c 40/10000/1000•
40      loop init for i:=2 to 1000 s(i):=true end loop init
c2 100/0/i•
100      j := j + i
run•
nc errors detected
2
3
5
7
11
13
*R002* user interrupt at line 150 in eratosthenes
edit•
edit mode
save erat•
member already exists -- really overwrite it? y•
list remote6•
sys•
system mode
attach•
session to attach to? ?•
    no  state mode line      program
    3   att  exec 015        inventory
    15  det  exec ---        custupd
    16  log  sys 008        ---
    23  att  edit 010        ---

session to attach to? 15•
elapsed time: 12 min 5 sec
cpu time: 0 min 23 sec
i/o activity: 0 transfers
session logging out

```

X001 inexistent customer at line 6820 in custupd
system mode
logout•
elapsed time: 48 min 26 sec
cpu time: 3 min 50 sec
i/o activity: 173082 transfers
session logging out

Appendix C-IV: Compile-time errors

=====

This appendix has to be completed as soon as all compile-time errors are known from the first PISA implementation. Thereafter, all examples of compile-time errors given in Section C have to be adapted to the correct error messages defined in this appendix.

Appendix C-V: Run-time errors

=====

This appendix defines the run-time errors that are not implementation dependent. These errors result from an error detected during execution of a PISA program in run mode (C-5.). As defined in C-5.3., the error location is reported immediately succeeding the run-time error message.

R001 <stop text>

The predefined procedure "stop" has been invoked (cf. B-10.6.). <stop text> is the string passed as actual parameter to the procedure "stop".

R002 user interrupt

The user entered a user interrupt during the time the user interrupt facility was enabled.

This appendix has to be completed as soon as all run-time errors are known from the first PISA implementation. Thereafter, all examples of run-time errors given in Section C have to be adapted to the correct error messages as defined in this appendix.

Appendix C-VI: Collected PISA command syntax

System_mode

```

system_cmd      ::= "edit•" | prog_name [ " " comm_string ] "•"
prog_name       ::= member
comm_string     ::= {character}
member          ::= implementation_dependent_member_name_syntax

```

Edit_mode

```

edit_cmd        ::= "sys•" | add_one_line | add_cmd |
                  include_cmd | delete_cmd | change_cmd |
                  renumber_cmd | print_cmd | verify_cmd |
                  list_cmd | save_cmd | compile_cmd |
                  translate_cmd | check_cmd | run_cmd

add_one_line    ::= line_number ( " " | tabulation-character )
                  {character}
line_number     ::= digit {digit}

add_cmd         ::= "a " first_line " " increment "•"
first_line      ::= line_number | "+"
increment       ::= digit {digit}

include_cmd     ::= "i " first_line " " increment " " member "•"

delete_cmd      ::= "d " range "•"
range           ::= line_number [ ".." line_number ] | "*"

change_cmd      ::= (change_all | change_n-th) range
                  delim old_text delim new_text "•"
change_all      ::= "c "
change_n-th     ::= "c" digit [ digit ] " "
delim           ::= character
old_text        ::= character {character}
new_text        ::= {character}

renumber_cmd    ::= "r " line_number " " increment "•"

print_cmd       ::= "p " range "•"

verify_cmd      ::= "v " range delim substring "•"
substring       ::= character {character}

```

```

list_cmd      ::= "list " routing [ " " options] "•"
routing       ::= "term" | device_address
options       ::= implementation_dependent_list_options

save_cmd      ::= "save " member "•"

compile_cmd   ::= "compile" [ " s] "•"

translate_cmd ::= "translate" [ " s"] "•"

check_cmd     ::= "check" [ " s"] "•"

run_cmd       ::= "run" [ " s] [ " := " comm_string] "•"

```

Halt_mode

```

halt_cmd      ::= "sys•" | "edit•" | "cont•" | "loc•" |
                  {simple_stmt | struc_stmt}
simple_stmt    ::= (* see B-6.2. *)
struc_stmt    ::= (* see B-6.3. *)

```


Appendix C-VII: Keyword index

=====

account	C-1.3.
add a single line	C-4.3.1.
add command	C-4.3.2.
attach	C-3.2.3.
attached state	C-2.3.
backspace	C-1.2.3.
C-code, C-code interpreter	C-5.2.
C-library	C-1.5.
change command	C-4.3.5.
check command	C-4.9.
compile command	C-4.7.
compile-time errors	C-4.7.
continue command	C-7.2.3.
delete command	C-4.3.4.
delete input line	C-1.2.2.
detached state	C-2.4.
edit mode	C-4.
edited text	C-4.2.
end-of-line characters	C-1.2.1.
execute errors	C-6.3.
execute mode	C-6.
halt mode	C-7.
idle state	C-2.1.
immediate statements	C-7.3.
include command	C-4.3.3.
input line	C-1.2.1.
interrupt	C-1.2.4.
interruption location	C-5.4.
libraries	C-1.5.
line numbers	C-4.2.1.
list command	C-4.5.
list directive statement	C-4.2.2.
location command	C-7.2.4.
login program	C-3.2.1.
login state	C-2.2.
logout program	C-3.2.2.

member	C-1.5.
print command	C-4.4.1.
privileges	C-1.3.
prompter	C-1.2.5.
range of text lines	C-4.2.4.
renumber command	C-4.3.6.
run command	C-4.10.
run mode	C-5.
run-time errors	C-5.3./C-5.4.1.
S-library	C-1.5.
save command	C-4.6.
short error messages	C-4.7.
stall output	C-1.2.3.
state of session	C-2.
stop procedure	C-5.4.2.
system mode	C-3.
T-code	C-6.2.
T-library	C-1.5.
term program	C-3.2.4.
terminal i/o	C-1.2.1.
text area	C-4.2.1.
text line	C-4.2.1.
translate command	C-4.8.
user interrupt	C-1.2.4./C-5.4.3.
utility programs	C-3.3.
verify command	C-4.4.2.

SECTION D
-----Applications of PISA

This section discusses PISA's use in several typical classes of applications. It does not define any properties of the PISA system but merely serves to support the understanding of PISA and to clarify its design guidelines. Since the role of an algorithmic language in application programming is well understood and not vastly different from language to language, the main weight of this section lies on describing the use of PISA's interfaces and of the PISA system as a whole. Besides this, the economy of interactive software production is discussed in a few paragraphs.

Table of contents

BATCH USAGE OF PISA	D-1.
FILE ACCESS	D-2.
The operation-oriented approach	D-2.1.
The file-oriented approach	D-2.2.
Data base systems	D-2.3.
EDUCATION	D-3.
COMMEPCIAL APPLICATION SOFTWARE	D-4.
The economy of interactive software production	D-4.1.
Software production with PISA	D-4.2.
Interactive software	D-4.3.
Batch software	D-4.4.
LANGUAGE FRONT END PROCESSORS	D-5.

D-1. BATCH USAGE OF PISA

=====

The reader who followed the description of the PISA system to this point noted that PISA's fundamental design is based on its use as an interactive programming system. Its user maintains a real-time dialogue with PISA by operating a terminal connected to the PISA system. Unless an executing PISA program explicitly detaches itself from the terminal, the connection is preserved during the creation, the test, and the execution of a PISA program. This "interactive" mode of using PISA has been defined in Section C; PISA's application in a batch environment is shown in this chapter.

If the PISA system has to be used in batch mode, the terminal interaction is simulated by a special PISA program, called "batch processor" in the sequel. The batch processor controls a PISA session via a pseudo terminal (cf. SSR 2000 to SSR 2004 in Appendix B-II and C-1.1.). It accepts input from any sequential file (presumably cards), enters it on the pseudo terminal, retrieves the output from the pseudo terminal, and produces a sequential output file (presumably a printfile). The batch processor itself may run in the "detached state" (C-2.4.), or it can be connected to a terminal. It may be designed to get accounting information from the controlled session, to detach and reattach controlled sessions, to handle several controlled sessions simultaneously, etc.

The most basic version of a batch processor is one that performs the following steps:

- (1) Create a pseudo terminal with SSR 2000.
- (2) Wait until the pseudo terminal requests input or has produced output to read (SSR 2004).
- (3) If the pseudo terminal waits for input, then go to (4), else go to (7).
- (4) Get next input record. If input reached end-of-file, go to step (5), else go to step (6).
- (5) Make sure that the controlled session is logged out, delete the pseudo terminal with SSR 2001, and terminate the batch processor.

- (6) Strip any trailing blanks from the input record, append an end-of-line to it, and enter it on the pseudo terminal with SSR 2002.
- (7) Retrieve the output from the pseudo terminal with SSF 2003 and write it to the output file, then go to step (2).

Obviously, such a batch processor neither performs any checks on the input, nor inspects the output retrieved from the pseudo terminal. As a consequence, the PISA user supplying input records to the batch processor anticipates a specific dialogue between the batch processor and the controlled session and sets up his input accordingly. If the dialogue is not exactly the one he expected, e.g. because of any unforeseen errors, the input might quite easily get "out of step" with the dialogue. If several users run their jobs in the same stream of input to the batch processor, a user could be penalized by the errors of one of his predecessors for he does not start his session at a well defined state.

A more practicable batch processor, although being based on the same principle of operation, accepts a certain simple command language and translates it to PISA session commands which are entered on the pseudo terminal. On the output side of the dialogue, such a batch processor examines the data received from the pseudo terminal and passes it on to the output file in a suitable form. If it detects an error, it recovers the controlled session to a well defined state and continues processing of the input at an appropriate point.

D-2. FILE ACCESS

=====

As introduced in B-11., a PISA program has two interfaces to communicate with its environment: the "user interface" to a terminal and the "system interface" to the run-time system (or operating system). Chapter B-11.2. described the system service requests (SSRs) used to pass and receive data through the system interface. Since file storage and input/output operations are performed and controlled by the run-time system, a PISA program accesses all files via the system interface.

The file access methods available to a PISA program depend on the particular PISA implementation. It most likely supports the access methods that the underlying operating system provides. Since not even basic access methods have a standard definition uniformly implemented in all operating systems, and because the set of access methods supported is not consistent among all operating systems, PISA does not define portable SSRs for file access. In practice, however, we notice that at least sequential and direct access to files is possible in almost every operating system and that these access methods do not differ too much from operating system to operating system. "Direct access" here means random access to any existing record in the file, where the record is designated by its ordinal number within the file.

Every PISA implementation supports the semiportable SSRs for sequential access to files (see SSRs 4010 to 4012 in Appendix B-II) and the ones for direct access to files (see SSRs 4020 to 4024 in Appendix B-II). Other file access methods like index-sequential, library-access, special random-access, etc., have to be supported with nonportable system service requests.

The following three chapters describe the usage of PISA in three different approaches to file processing.

D-2.1. The operation-oriented approach

The classical approach to file processing is what we may call "operation-oriented". For each file access method, a set of file operations (open, close, read, write, etc.) is defined. The program invokes the desired operation routine and passes the identification of the file as a parameter. The common and well known sequence of file operations is to first open the file, then

to process the records, and finally to close the file. We refer to this scheme of file processing as "operation-oriented" because the program issues a file operation and specifies on what file the operation is to be performed. In the "file-oriented" approach, discussed in D-2.2., the program invokes a routine dedicated to a specific file and specifies what operation is to be performed on this file.

It is expected that a PISA installation provide external routines for performing all file operations. These routines may be adapted to the local needs. The following three external routines are examples of very basic file operation routines on sequential disk files. They are not intended to show a practical implementation of such routines since they particularly do a bad job on error handling and are restricted to disk files. The purpose of the routines is to give an idea on how operation-oriented file access can be programmed in PISA.

```
external procedure operseq (var fileno: 1..1000,
                           filename: string(8), mode: pm, area io_area)

type parm2 = record
    file_#:          1..1000
    comp_code:       0..10
    diagn:           string(30) vary
end record
type parm3 = record
    filename:        string(50) vary
    proc_mode:       0..2
    shared_acc:      boolean
    reclen:          1..100000
    recperblock:     0..10000
    cluster:         0..100000
    device_cntl:     string(20) vary
end record
type pm =            (inp,out,upd)

var p2:              parm2
var p3:              parm3

p3 := parm3(:filename,ord(pm,mode),false,areazize(io_area),
            0,0,':')
sys(4010,p2,p3) (* open file *)
```



```

case p2.comp_code
  when 0: fileno := p2.file_#
  when 1: stop('file not found')
  when 2: stop('file already exists')
  when 3: stop('protection violation')
  when 4: stop('file in use')
  when 5: stop('illegal filename')
  otherwise: stop('open error: ' + p2.diagn)
end case
end procedure openseq

```

```

external procedure readseq (fileno: 1..1000, area io_area,
  var eof: boolean)

```

```

var p2: record
  file_#:          1..1000
  comp_code:       0..10
  diagn:           string(30) vary
end record

```

```

p2.file_# := fileno
sys(4012,p2,io_area)  (* read next record *)
case p2.comp_code
  when 0: eof := false
  when 1: eof := true
  otherwise: stop('read error: ' + p2.diagn)
end case
end procedure readseq

```

```

external procedure closeseq (fileno: 1..1000)

```

```

var p2: record
  file_#:          1..1000
  comp_code:       0..10
  diagn:           string(30) vary
end record

```

```

p2.file_# := fileno
sys(4011,p2)  (* close file *)
if p2.comp_code<>0 then
  stop('close error: ' + p2.diagn)
end if
end procedure closeseq

```

As mentioned before, these routines are more complex in practice. Presumably, they are coded and tested by an experienced programmer before they are stored on the system C- and T-libraries and thus made available to application programmers.

The main procedure shown below is an example of a very simple PISA program that makes use of the operation-oriented routines introduced above. The program's purpose is to print a list of all customers.

main procedure cust_list

```

var customer: record
    #:          1..100000
    name:       string(30)
    rest:       string(66)
end record
var cust_file: 1..1000
var eof:       boolean
type pm =      (inp,out,upd)

procedure openseq external (var 1..1000,string(8),pm,area)
procedure readseq external (1..1000,area,var boolean)
procedure closeseq external (1..1000)

openseq(cust_file,'cmaster',inp,customer)
loop produce_list
    readseq (cust_file,customer,eof)
    if eof then exit produce_list end if
    print (edit(customer.#,'ZZZZZ9 '),customer.name)
end loop produce_list
closeseq(cust_file)
end procedure cust_list

```

The declarations of the external procedures needed for a certain access method could be stored as a member in the system S-library. They then could be copied into the PISA program by an include command (C-4.3.3.) as e.g.,

```
i + 10 seq_acc
```

D-2.2. The file-oriented approach

In what we call the "file-oriented" approach to file processing, the program does no longer invoke routines that perform a certain operation on the file given as actual parameter. All operations on a specific file are performed by one single external routine. The program invokes the routine dedicated to the file to process and specifies the requested operation.

The following example of a "file-oriented" routine processes the customer file introduced in the program "cust_list" above. It simulates direct access to the sequential customer file, i.e. the program using this routine virtually processes a direct file. The only restriction compared to a physical direct access file is the key values not being allowed to ever decrease. The customer file may be opened as input or as update file. In case of input, the records can only be read, otherwise they can be read, new ones can be added to the file, and existing ones can be rewritten or deleted.

```
external coroutine procedure cust_direct (op: optype,
                                         var io_area: cust_rec, var found: boolean)

type cust_rec = record
    #:                1..100000
    name:            string(30)
    rest:            string(66)
end record

type pm             = (inp,out,upd)
type optype         = (openinp,openupd,read,write,delete,close)

var opened_as:      optype
var input_rec:      cust_rec
var input,output:   1..1000
var last_output_#:  1..100000
var eof:            boolean

procedure openseq   external (var 1..1000,string(8),pm,area)
procedure readseq   external (1..1000,area,var boolean)
procedure writeseq  external (1..1000,area)
procedure closeseq  external (1..1000)
procedure delete    external (string(8))
procedure rename    external (string(8),string(8))
```

```

if op<>openinp and op<>openupd then
  stop('no open on customer file')
end if
opened_as := op
openseq(input,'cmaster',inp,input_rec)
if op=openupd then openseq(output,'cmnew',out,io_area) end if
readseq(input,input_rec,eof)
last_output_# := 0
suspend

loop process_file while op<>close
  if op=openinp or op=openupd then
    stop('invalid file op')
  end if
  loop search_record while not eof and input_rec.##<io_area.##
    writeseq(output,input_rec)
    last_output_# := input_rec.##
    readseq(input,input_rec,eof)
  end loop search_record
  found := not eof and input_rec.##=io_area.##
  if op=read then
    if found then io_area := input_rec end if
  else
    if opened_as<>openupd then stop('invalid file op') end if
    if op=write then
      if io_area.##<=last_output_# then
        stop('invalid key sequence')
      end if
      last_output_# := io_area.##
      writeseq(output,io_area)
    end if
    if found then readseq(input,input_rec,eof) end if
  end if
  suspend
end loop process_file

loop copy_rest while not eof
  writeseq(output,input_rec)
  readseq(input,input_rec,eof)
end

closeseq(input)
closeseq(output)
if opened_as=openupd then
  delete('cmaster')
  rename('cmnew','cmaster')
end if

end procedure cust_direct

```

For the sake of simplicity, the above routine makes use of the external routines "openseq", "closeseq", and "readseq" introduced in D-2.1. The external routine "writeseq" is defined similarly for writing a record to the sequential file specified; the external routines "delete" and "rename" perform the operation suggested by their names.

"File-oriented" access routines can considerably simplify programs processing sequential files because these programs access the file on a logical rather than on a physical basis. The program structure of a file update program, for example, remains the same regardless of whether the updated file is a sequential or a random access file. Besides simplification, we gain device independence and portability since a program does no longer rely on a particular physical implementation of the files it accesses.

Although it is not the thesis of this report, it is worth mentioning that the "file-oriented" approach to file access allows the implementation of sophisticated file management systems:

- Accounting data can be updated based on specific file usage.
- Statistics on file usage may be maintained by the file access routines. These statistics could then, for example, be used to automatically select another storage medium or access method for the file if its present implementation turns out to be inefficient.
- If all parameters of files (recordsize, blockfactor, volume, external name, etc.) are stored in a table on secondary storage, the "file-oriented" access routines can access this table, use its data for opening and processing the file, and update the table if necessary. Such a file management system allows to manipulate files rather flexibly without changing anything at the software accessing the files.
- Control of shared access to files may be programmed into the file access routines if the operating system does not perform the desired checks.

As the example given for simulated direct access to a sequential file has shown, "file-oriented" access routines can be implemented straightforward with PISA's coroutines.

D-2.3. Data base systems

The interface between PISA programs and data base software has to be implemented with nonportable system service requests. Similar to conventional file accesses as outlined in the previous two chapters, the data base system service requests are presumably not programmed by the common PISA programmer. They are packed into external procedures or functions coded and tested by senior programmers.

D-3. EDUCATION

=====

Apart from efficiency considerations, the main requirements to a programming language for introductory computer science courses are consistency and simplicity. "Consistency" means that the language's concept should be consistent with the concept of data abstraction and algorithmic structuring methods taught. The students should not have to perform a mental transformation of the "model of computation" they finally grasp into yet another model, the programming language. The "simplicity" of a programming language is reflected by the time it takes to learn its syntax and semantics. A student should be able to completely understand the language he uses, and this knowledge should result as a by-product of the course, not as its main topic. If the student does not have to struggle with weird syntax rules and obscure semantics, he can concentrate on principles rather than on nasty details of their implementation.

Since PISA is not a language exclusively devoted to education, it includes certain features that are not necessarily needed for teaching, at least not in basic courses. These features (like separate compilation, decision tables, the system interface, coroutines, gotos and labels, etc.) can, however, be left away from the language subset taught; they have no effect on the rest of the language. Such a PISA subset comes quite close to Pascal, except for some minor syntactic variations and the different definition of the block structure. Based on the wide acceptance of Pascal, we can assume that PISA could be used as a reasonable tool in education. It is consistent with common data abstraction models and offers the usual control structures. The PISA subset required for a basic programming course can be acquired in quite a short time.

Experience shows that beginner students usually have considerable difficulties in programming input/output on one hand and in finding the reason for program malfunctions on the other hand. In a basic computer science course with PISA, the students would program all input and output using the predefined procedures for handling the PISA user interface (print, input, etc.). These procedures are very flexible (see B-11.1.) yet easy to understand because they do not introduce additional syntax rules and have simple semantics. The conversion rules defined for the input and print procedures are closely related to the PISA language definition: an input to PISA must be a valid constant of the receiving variable's type (literal strings not enclosed in quotes, of course); a value to print out is converted to string

representation with the predefined conversion function "str", thus resulting in a valid constant of the output value's type (cf. E-10.4.).

The definition of the PISA session is the base of the method for investigating program malfunctions. A program is automatically switched from run mode to halt mode after a run-time error. The student can also switch it manually to halt mode if he detects an abnormal program behaviour. In halt mode, he may enter PISA statements for immediate execution or halt commands to continue or abort the program's execution. The predefined procedure "trace" is of special utility to trace the program flow; it allows fast, slow, and single-step trace. The advantage of immediate execution of PISA statements entered during halt mode is that the student does not have to learn a special command syntax to use the interactive run-time system. Other interactive languages as APL and BASIC have already proven the benefits of such a dialogue oriented programming system.

The conception of an interactive use of PISA in education relies on sufficient hardware resources like terminals, processor time, memory space, and secondary storage space. At the first glance, the cost of the resources required might seem unacceptably high. But if we found a cost calculation on 1978's prices for minicomputers (e.g. PDP-11), secondary storage devices, and terminals, and if we further assume a nonprofit-oriented pricing policy for an average terminal use of 100 hours per month, we get a rate of approximately 10 US-\$ per hour connect time. This figure includes a depreciation of 10 to 15 percent per year. If we trust in the hardware cost predictions for the next decade, we can estimate an hourly rate of roughly 2 to 4 US-\$ for the same resources by 1985. These figures do not seem too high for the obvious advantages of an educational interactive programming system.

D-4. COMMERCIAL APPLICATION SOFTWARE

=====

PISA is primarily designed as a tool for producing application software in a commercial environment. The term "commercial" is used in opposition to "educational"; the term "application software" in opposition to "system software". In an educational environment, programs are written and tested for training purposes; maintainability, portability, compatibility with existing data bases, efficiency, etc., are of minor interest. Usually, student programs are discarded anyway once they run properly. Commercial software, on the other hand, is generally produced for medium to long term usage and has to be created and maintained under a time and money constraint.

This chapter discusses some topics of commercial programming with the PISA system.

D-4.1. The economy of interactive software production

PISA has been designed in the author's strong belief in interactive software production. Interactive software creation and testing has only one disadvantage over batch programming and testing: it requires more hardware resources and more sophisticated system software. Compared to this disadvantage, the interactive approach has a number of advantages over batch that are of paramount importance:

- Entering and modifying source code on a terminal is more convenient, done faster, and less error-prone than handling bulky card decks or punching source library updates.
- The turnaround time of a job is cut down to its execution time. The output files produced by the job can be inspected immediately from the terminal.
- After a program produces a run-time error or is interrupted from the terminal, the program retains its state and halts at the interruption point. All objects accessible at this point can be denoted in statements entered for immediate execution. Errors can be located easier than with a post-mortem dump. If the error allows, the erroneous values may be corrected and program execution resumed at the interruption point.

- Since all relevant data may be inspected from the terminal, considerably less printout is produced during software production and maintenance.

The danger of interactive software production is the challenge to start testing a program before it has been sufficiently examined and verified at the desk. Proper program design and desk tests are issues that must be taught in education and enforced by project management. An interactive system does in no way simplify these tasks.

If a programmer is taught how to create and test programs with an interactive system, he will be much more productive with this interactive system than with a batch system. If this increase in productivity is worth more than the difference between the interactive and the batch programming system's hardware cost, an interactive system should be chosen in the long term. Hardware prices and price predictions as discussed in D-3. allow us to forecast that the number of commercial programming projects where the interactive approach is cheaper than the batch approach will be steadily growing.

D-4.2. Software production with PISA

The definition of the PISA session (Section C) is based on a certain strategy of software production. This strategy is described below by decomposing it into a number of steps and showing the use of PISA in each step. Notice that the term "software production" is interpreted in its purest form here: the creation of software (or programs) to be executed by given hardware. We do not treat issues like problem description, documentation, and project management but are well aware that they are part of a software production strategy as well. So let us concentrate on the production of programs.

- (1) Based on the problem specification, a PISA program is coded. Existing external routines and precoded pieces of PISA source should be used wherever they ease the task of the programmer; the PISA programmer should not have to code "sys" procedure invocations. The PISA system is not involved in this step.
- (2) The coded program is entered on a terminal either in PISA edit mode (cf. C-4.) or with any other suitable editor. After the program has been entered, it can be immediately checked for syntax errors if desired. At the end of this

step, the programmer requests a printout of the program. This step uses only minor resources of the PISA system, particularly not much processor time.

- (3) The programmer checks and verifies the program at his desk. He corrects the source program at a terminal and, if necessary, requests new printouts of the program. After he expects the program to execute correctly, he prepares test data. The PISA system mode and the edit mode are used during this step.
- (4) The program is tested in run mode and halt mode. Quite likely, several recompilations are done during this step. Except for the execute mode, the full PISA system is used in this step. A relatively high memory space and processor time occupation has to be expected during program testing.
- (5) Once the program is performing as expected, it is compiled to C-code (slow code) and stored in the proper C-library.
- (6) For a certain while after the program has been released, its C-code is used for "real life" runs. This facilitates to find the reason for run-time errors since the program is switched to halt mode after a run-time error. During this initial period, the program consumes much processor time since the execution of the C-code performs all checks to catch run-time errors as soon as possible (cf. C-5.2.).
- (7) Following a suitable time of error-free program runs, the program is translated to fast T-code (cf. C-6.2.). From then on, the processor time used per average run decreases significantly.

D-4.3. Interactive software

A common interactive PISA program processes a dialogue with one single terminal that is connected to the program's user interface (cf. B-11.1.). The predefined procedures and functions for handling the user interface have been defined in B-11.1. They provide a standard set of operations that can be performed on the user interface, i.e., on the terminal. If additional operations as cursor positioning, video control, clearing all or part of the screen, etc., are required, they should be programmed in external

procedures or functions rather than in the program itself. These operations have not been included in the predefined routines because they, unfortunately, are not standardized. Packing them into external routines makes the program terminal-independent; a cursor positioning could, for example, be requested by the procedure statement "pos(x,y)", where x and y specify the coordinates of the cursor position. The external procedure sends the appropriate characters to the terminal with "printr(...)" .

A user of interactive application software should not be involved in the command syntax of the system supporting the software. The definition of PISA's system mode (C-3.) showed that the login program is automatically run when input is entered on a terminal connected to a logged out session. Since the login program is available in source code on every installation, it can implement a problem oriented login dialogue and finally chain to the appropriate program. If more than one program must be run for a certain application, the "chain to program" SSF can be used to pass control from one program to another without any user interaction. At end of the session, a program may issue a "logout" SSR or chain to the standard logout program. Thus, the user of a PISA program can be relieved of any session control dialogue.

If a PISA program has to support a terminal that cannot be connected to the user interface because of its operation characteristics (e.g. block-mode transfer), nonportable system service requests must be used to access the terminal. Non-portable system service requests are also required for multi-terminal processing by a single PISA program.

D-4.4. Batch software

Although interactive software is economically used for more and more applications, there will always be a significant number of problems that can be solved cheaper with batch software. Hence, it is quite worthwhile to have a look on how PISA could be used for batch software.

Conventionally, a part of the operating system schedules and controls programs to be executed in batch mode. The user specifies the desired actions to be performed with a job control language that is interpreted by the batch job scheduler. PISA's batch use is based on the principle that a PISA program must be able to request all services from the operating system

exclusively via SSRs so that traditional job control languages become obsolete.

A stream of batch type PISA programs can be started from a terminal. The first program detaches the session from the terminal (C-2.4.). The programs then chain from one to another directly or each may chain back to some controlling program that selects the next one to execute. The communication string provides a convenient means of passing short information strings between programs.

Another way of using the PISA system for batch type applications has been introduced in D-1. It makes necessary a PISA program acting as "batch processor". The batch processor interprets some batch command language and controls one or more PISA sessions via a pseudo interface. If such a batch processor executes all day long, it could inspect a "batch schedule table" periodically and automatically start batch runs at a given time.

D-5. LANGUAGE FRONT END PROCESSORS

=====

The use of high-level programming languages during the past has shown that there frequently arises the need to support language extensions to what is called the "host-language". Examples of such extensions are macro commands, simulation constructs, decision tables, nonprocedural problem-oriented code structuring, etc. Conventionally, such language extensions are processed by a so called "preprocessor" or "translator" which produces an output text in the host-language. This output text is then passed to the host-language compiler, resulting in an executable program. The drawbacks of this strategy and its inapplicability for interactive software development have been shown in A-1.3.

If a preprocessor or translator to PISA is written, the following guidelines should be observed whenever feasible:

- The user should still have available all the flexibility of interactive software creation, testing, and usage.
- The user should not be concerned with the translation of the language extensions. All diagnostic messages he gets should be related to his input text (the one with the extensions), not to the generated PISA program.

To stress the difference between conventional "batch-type" preprocessors and such interactive preprocessors, we call the latter "language front end processors" (see also A-1.3.).

A language front end processor to PISA is presumably a PISA program interacting with the user via the user interface on one side and controlling a PISA session via a pseudo terminal on the other side. It accepts input from the user and translates it to valid PISA program text which is entered on the pseudo terminal. The controlled session is requested to compile and run the PISA program after the input has been completed. The language front end processor retains the mapping of the input text to the generated PISA program. This allows it to analyse the error messages (and possibly tracing data) received from the pseudo terminal, to relate them to the original input text, and to output appropriate diagnostic messages to the user.

The implementation of front end processors to PISA was the main reason for including a goto statement and label declarations in the language PISA. Some language extensions can be translated much easier into "goto text" than into "if-then-else / case / loop text". This is particularly evident for translation of nonprocedural extensions.

SECTION E

Implementation Guidelines

Theoretically, the definition of the programming language PISA, together with the description of the PISA session, is a complete reference for a PISA implementation. Nevertheless, it seems quite worth-while giving some guidelines to a PISA implementor. Part of the reason for this is the fact that several constructs of PISA have been carefully designed to meet specific requirements like efficient implementation, testing support, consistency with existing data, etc. The implications of some PISA features might not be apparent to an implementor, at least not at an early stage of the implementation work. One of the purposes of this section can be seen in the trial to narrow the information gap between the designer and the implementor.

Another goal of the following chapters is to present some of the work done during the design of PISA that can be of benefit to an implementor by providing him a basis to start his implementation work from. The SLR(1) parser grammar for PISA is one example for this, the implementation of coroutines another.

Table of contents

THE COMPILER	E-1.
An SLR(1) grammar for PISA	E-1.1.
Partial recompilation	E-1.2.
EXECUTION OF A PISA PROGRAM	E-2.
The C-code	E-2.1.
The T-code	E-2.2.
Representation of data types	E-2.3.
Evaluation of string expressions	E-2.4.
Coroutine run-time management	E-2.5.
Relation tests on real operands	E-2.6.
THE USER INTERFACE	E-3.
The input device	E-3.1.
The output device	E-3.2.
The terminal handler	E-3.3.
The input processor	E-3.3.1.
The output processor	E-3.3.2.
The terminal handler control table	E-3.3.3.
THE SYSTEM INTERFACE	E-4.
Appendix E-I: Keyword Index	
Appendix E-II: References	

E-1. THE COMPILER

=====

The method chosen for writing a compiler is largely a matter of personal preference and of the software tools available. No recommendations for a specific type of PISA compiler are given here. To help the construction of a parser, an SLR(1) grammar for PISA is shown in E-1.1. This does by no means imply that PISA should be parsed bottom-up; a deterministic top-down parser with a one symbol lookahead can parse a PISA program as well. Chapter E-1.2. describes some guidelines for partial recompilation of PISA programs.

E-1.1. An SLR(1) grammar for PISA

The grammar given in Section B serves as a "user grammar". The following SLR(1) grammar [DeFemer,71] for the language PISA is designed to support the production of a parser for PISA.

The grammar below does not include productions for the lexical structure of PISA (cf. B-1.). Therefore, the following syntactic constructs appear as terminals in it:

<id>	= identifier
<integer>	= integer number
<number>	= real number
<litstring>	= literal string
<condentry>	= condition entry
<actentry>	= action entry

The grammar of all these syntactic constructs has been defined in B-1. In a practical parser implementation, they are presumably parsed by the scanner and returned to the parser as one syntactic entity.

For better legibility, the word symbols of PISA are underlined in the SLR(1) grammar. <empty> denotes the null sequence of symbols. The goal symbol is <compilation>.

Procedures and functions

```

<compilation> ::= main <procdef>
                | external <procdef>
                | external <funcdef>

<procdef>      ::= coroutine <simpleproc>
                | <simpleproc>

<simpleproc>    ::= procedure <id> <fparm>
                  <block> end procedure <id>

<funcdef>      ::= coroutine <simplefunc>
                | <simplefunc>

<simplefunc>    ::= function <id> <fparm> : <type>
                  <block> end function <id>

<fparm>        ::= ( <fplist> )
                | <empty>

<fplist>       ::= <fplist> , <fparmdcl>
                | <fparmdcl>

<fparmdcl>     ::= area <id>
                | var <id> : <type>
                | <id> : <type>

```

Blocks

```

<block>        ::= <dclpart> <stmtpart>

<dclpart>      ::= <dclpart> <dcl>
                | <empty>

<stmtpart>     ::= <stmtpart> <stmt>
                | <empty>

```

Declarations

```

<dcl> ::= type <id> = <type>
      | const <id> = <constdef>
      | var <idlist> : <type>
      | bind <id> to <reference>
      | procedure <id> external <ptypelist>
      | function <id> external <ptypelist> : <type>
      | procedure <id> forward <ptypelist>
      | function <id> forward <ptypelist> : <type>
      | <procdef>
      | <funcdef>

<idlist> ::= <idlist> , <id>
           | <id>

<ptypelist> ::= ( <ptlist> )
              | <empty>

<ptlist> ::= <ptlist> , <typedcl>
           | <typedcl>

<typedcl> ::= area
            | var <type>
            | <type>

```

Data types

```

<type> ::= <id>
        | <typedcl>

<typedcl> ::= ( <idlist> )
            | <lowbound> .. <const>
            | fix ( <const> .. <const> , <const> )
            | float ( <const> .. <const> , <const> , <const> )
            | string ( <const> ) vary
            | string ( <const> )
            | -> <id>
            | packed <structype>
            | <structype>

<lowbound> ::= <addop> <litconst>
            | <addop> <id>
            | <litconst>
            | <id>

```

```

<structype>      ::= array ( <typelist> ) of <type>
                  | set of <type>
                  | record <fielddcl> <varpart> end record

<typelist>       ::= <typelist> , <type>
                  | <type>

<fielddcl>       ::= <fielddcl> <idlist> : <type>
                  | <empty>

<varpart>        ::= case <id> <whenvnt> <otherwvnt> end case
                  | <empty>

<whenvnt>        ::= <whenvnt> when <whenlist> : <fielddcl>
                  | <empty>

<otherwvnt>      ::= otherwise : <fielddcl>
                  | <empty>

<whenlist>       ::= <whenlist> , <whenval>
                  | <whenval>

<whenval>        ::= <const>
                  | <const> .. <const>

```

Constants

```

<constdef>      ::= <addop> <litconst>
                  | <addop> <id>
                  | <litconst>
                  | <id>
                  | <id> <complist>
                  | <typedcl> <complist>

<const>         ::= <addop> <litconst>
                  | <addop> <id>
                  | <litconst>
                  | <id>

<litconst>      ::= <integer>
                  | <number>
                  | <litstring>
                  | nil

```

References to constants, variables, procedures, and functions

```

<reference>      ::= <id>
                  | <reference> <substr>
                  | <reference> ( <exprlist> )
                  | <reference> . <id>
                  | <reference> ->

<substr>         ::= <: <expr> :>
                  | <: <expr> , <expr> :>

<exprlist>       ::= <exprlist> , <expr>
                  | <expr>

```

Statements

```

<stmt>           ::= <labeldcl> <unlabstmt>

<labeldcl>       ::= <labeldcl> * <idlist> :
                  | <empty>

<unlabstmt>      ::= <reference> := <expr>
                  | <reference>
                  | goto <id>
                  | exit <id>
                  | suspend
                  | begin <id> <block> end begin <id>
                  | if <expr> then <block> <elsepart> end if
                  | case <expr> <whenpart> <otherwpart> end case
                  | datab <condpart> <actpart> end datab
                  | loop <id> <loopcontr> <block> end loop <id>

<elsepart>       ::= else <block>
                  | <empty>

<whenpart>       ::= <whenpart> when <whenlist> : <block>
                  | <empty>

<otherwpart>     ::= otherwise : <block>
                  | <empty>

<condpart>       ::= <condpart> <cond>
                  | <cond>

<cond>           ::= cond <expr> <condentry>

<actpart>        ::= <actpart> <action>
                  | <action>

```

```

<action>      ::= action <actentry> <block>

<lcopecntr>   ::= for <id> := <expr> <direction> <expr>
                  | while <expr>
                  | <empty>

<direction>   ::= to
                  | downto

```

Expressions

```

<expr>        ::= <expr> or <conjunction>
                  | <conjunction>

<conjunction> ::= <conjunction> and <negation>
                  | <negation>

<negation>    ::= not <relation>
                  | <relation>

<relation>    ::= <sum> <relop> <sum>
                  | <sum>

<sum>         ::= <sum> <addop> <term>
                  | <term>

<term>        ::= <term> <multop> <factor>
                  | <factor>

<factor>      ::= <factor> ** <simpfactor>
                  | <simpfactor>

<simpfactor>  ::= <reference>
                  | <litconst>
                  | ( <expr> )
                  | <addop> <simpfactor>
                  | <id> <complist>

<complist>    ::= (: :)
                  | (: <clist> :)

<clist>       ::= <clist> , <comp>
                  | <comp>

<comp>        ::= <expr>
                  | <expr> .. <expr>
                  | <complist>

```

```

<relop>      ::= in
               | rep <compop>
               | <compop>

<compop>     ::= =
               | <>
               | <
               | >
               | <=
               | >=

<addop>      ::= +
               | -

<multop>     ::= *
               | /
               | div
               | mod

```

Remarks

The above grammar has been developed with the SLR(1) option of the LALR(k) parser generator of LaLonde [LaLonde et al.,71]. It contains:

- 166 productions
- 78 terminal symbols
- 60 nonterminal symbols

The nonterminals <lowbound>, <constdef>, and <const> all produce the four sentential forms

```

<addop> <litstring>
<addop> <id>
<litstring>
<id>

```

If these four sentential forms were generated by a single nonterminal, an SLR(1) parsing conflict would result.

Notice that the sentential form

<reference> (<exprlist>)

generates denotations of indexed variables, function designators, and procedure statements. The code generator has to distinguish between these three cases based upon the type of the object referenced.

The nonterminal <complist> (component list) is referenced to describe the syntax of structured constants (cf. B-3.1.) and the syntax of value constructors (cf. B-4.1.). In the former case, the code generator has to verify that all components in the component list be constants, not expressions as are permitted in value constructors.

E-1.2. Partial recompilation

During the test of a PISA program, a significant number of recompilations is very likely to occur in a short period of time. Practical observation reveals that most of the changes made to a program after a thorough desk test concern the correction of minor oversights like missing initializations, misspelled identifiers, omitted statements, etc. The majority of such corrections has only a very local effect on the PISA program, especially if the correction does not affect a declaration. It, therefore, would be a waste of resources to completely recompile a PISA program every time its text is changed. The following paragraphs outline a method that could be used for efficient partial recompilation. Further detailed research work on this topic would be of great benefit for language design in general and for a PISA implementation in particular.

As Section C defined, a PISA program is always compiled in edit mode. During the first compilation after the edit mode has been entered from system mode, the compiler builds up a "block reference table". This table is essentially a rooted directed tree that maps the PISA program's static block structure to the edited text (cf. C-4.2.) and to the generated code. Each node of the tree (i.e. each block) points to the beginning and to the end of the PISA source text belonging to that block's declaration and statement parts, to the beginning and to the end of the code generated by this block, and to the symbol table section associated with the declarations of this block.

If changes are made to a previously compiled PISA program, the edit mode marks the changes in the block reference table. The "declaration flag" of the block is set when a change is made to the block's declaration part or to a label declaration within the block, the "statement flag" if its statement part is altered. The next compile request then inspects the block reference table and recompiles parts of the program only: if the "declaration flag" of a block is set, the entire block and all its nested blocks are recompiled; if the "statement flag" is set, only the statement part of the block has to be recompiled, where the code of nested blocks can simply be copied without recompilation. After a "renumber" command (cf. C-4.3.6.), the entire program has to be recompiled.

Such a method of partial recompilation is expected to save a considerable amount of processor time in an interactive programming system like PISA without increasing its overall complexity significantly. The savings in processor usage result in a better response time, particularly if several users are simultaneously recompiling PISA programs. In present interactive programming systems, the resources absorbed by compilations quite often lead to a bottleneck in the system performance that increases their response times beyond an acceptable limit.

E-2. EXECUTION OF A PISA PROGRAM

=====

As has been introduced in Section C, two executable codes are defined for PISA: C-code and T-code. Chapters E-2.1. and E-2.2. describe the guiding principles for designing these codes. Chapter E-2.3. gives certain rules for the internal representation of data types. E-2.4 explains how string expressions can be implemented efficiently based on the definition of the type string, the operations on strings, and the predefined string handling functions. Since the introduction of hierarchical coroutines in PISA makes necessary a run-time routine management that differs from the well known handling of simple procedures and functions, Chapter E-2.5. is intended to give a PISA implementor some support in this respect. E-2.6., finally, shows an optimization method for relation tests on real operands.

An executing PISA program communicates with its environment via the user interface and via the system interface. These interfaces are not treated in this chapter; E-3. and E-4. describe their implementation.

E-2.1. The C-code

The C-code (or compiled code) is an intermediate code produced by compilation of PISA program text. It is executed in run mode by the C-code interpreter. The C-code can also be referred to as "slow code" or "test code" as opposed to the T-code which is the "fast code" or "production code". The format of the C-code remains undefined; the PISA implementor may adapt it to the underlying hardware and system software.

The following guidelines, however, must be observed when the C-code and the C-code interpreter are designed:

1. The C-code interpreter does not assume a correct PISA program. The principle to obey during interpretation of the C-code is: if anything could possibly be incorrect, check it for correctness.
2. Any run-time error is caught as early as possible. The interruption location is set according to C-5.4.

3. The format of the run-time error messages must be consistent with the definitions of C-5.3. The error text must be related to the PISA program text that caused the error, not to the effect the error had to the hardware or run-time system (e.g. "referenced variable does not exist" instead of "addressing exception").
4. As defined in Section C, the PISA session is switched to halt mode after a run-time error has been detected. In halt mode, the user may enter PISA statements for immediate execution (cf. C-7.). This implies that the symbol table be stored together with the C-code and that the active scope of identifiers be known at the interruption point. To conserve memory space, the symbol table should be stored on disk during interpretation of C-code. The active scope of identifiers can be denoted by a "scope stack" per coroutine which is dynamically maintained during interpretation of the C-code. Notice that this stack needs only to be updated upon entry/exit of a block that contains declarations.

As these guidelines show, the C-code and the C-code interpreter are not trimmed for high execution speed and modest storage requirements but for comprehensive error detection and testing support.

Generally, it is of paramount importance that the designer of the C-code and of the C-code interpreter does not only base his design on PISA's language definition but also on the definition of the PISA session in Section C.

E-2.2. The T-code

The T-code (translated code) of a PISA program is produced by the "translate" command (C-4.8.). The format of the T-code is undefined; it can be any intermediate code or machine code. If it is an intermediate code, it is executed by the T-code interpreter; if it is machine code, its interpreter is the hard- or firmware of the system.

When the T-code and, if necessary, its interpreter are designed, the implementor is obliged to observe the following guidelines:

1. The T-code interpreter assumes a correct PISA program. The only checks issued during execution of the T-code are

the ones required to ensure safe operation of the entire system. These checks are usually performed by the hardware (memory protection, illegal operations, etc.) and by the operating system (file protection, shared access rights, invalid record sizes, etc.). The principle to obey during execution of T-code is: do not protect the PISA program from its own errors. Since the interpreter of the T-code assumes a correct program at most places where the C-code interpreter issues a check, an execution error might not be caught immediately or can even remain undetected. Examples for this are type adaptation failures, uninitialized variables, illegal substring denotations, etc.

2. The format of the execution error messages must be consistent with the definitions of C-6.3. This particularly implies that the error location (line number and routine identifier) has to be reported after an execution error. The error text must not necessarily be related to the PISA program text that caused the error; the implementation handbook may describe the reasons that can lead to specific execution errors.

These guidelines show that the T-code and its interpreter (software, firmware, or hardware) are trimmed for execution speed and explicitly not for comprehensive error detection and testing support. In opposition to the C-code, the T-code does not have to make any provisions for catching the program to a well defined state after an execution error, only an execution error message as described above is required thereafter. The T-code and its interpreter is considerably simpler than the C-code and the C-code interpreter.

E-2.3. Representation of data types

When the internal representation of PISA's data types is defined for an implementation, the implementor should bear in mind the following principles:

1. The values of the predefined enumerated type "integer" are the positive and negative integers in the mathematical sense. Integer subrange declarations should be accepted at least over the range -10^9 to $+10^9$. If the hardware does not provide a unique integer representation capable of holding this subrange, several integer representations

must be used depending on the subrange type declaration. On a PDP-11, for example, a word (16 bit) may be used for integer subranges within -32768 to +32767, where for subranges outside this bounds multiple word representation must be chosen.

2. In order for a PISA program to be able to process data bases established by non-PISA programs, the type "fix" should be implemented by decimal representation if the machine supports decimal arithmetic. If signed and unsigned decimal numbers are distinguished by the hardware (e.g. Burroughs B-3700), signed decimal numbers should be used for fixed point types defining negative values, unsigned ones if the lower bound of the declared range is zero or greater.
3. For "string" types with the vary option, the length information should be encoded in the same way as other languages for the implementation's target machine do. This ensures compatibility with existing files.
4. Packed structured types must not necessarily be represented in a compressed form if storage space conservation is not an issue and if unpacked structures allow to access all existing data bases. Nevertheless, the PISA compiler must enforce the few restrictions imposed on the use of components of packed structured types (cf. B-3.5., B-4.3., and B-6.2.2.).
5. According to B-2.3.3., a minimum of 256 elements in a set must be supported by every PISA implementation. A set S of base type t should be represented by a sequence of n bits (where $n = \text{ord}(t, \text{last}(t)) - \text{ord}(t, \text{first}(t)) + 1$) preceded by any number of insignificant zero bits. If the significant bits are numbered 0, 1, ..., $n-1$, then bit i is one if and only if x is in S and $\text{ord}(t, x) = i$.

E-2.4. Evaluation of string expressions

The most severe drawback of evaluating string expressions as defined for most languages supporting string handling (e.g. PL/I, BASIC) is that the length of the string temporaries cannot be determined at compile-time. Therefore, a rather complex dynamic space allocation scheme for string temporaries has to be implemented. In PISA, the so called target type of a string expression is always known at compile-time: it is the type of

the variable on the left hand side of a string assignment, or it is the type of the formal parameter if the string expression is an actual parameter.

Since no formal parameter declaration is given for predefined procedures and functions, the target type of an actual string parameter to a predefined procedure or function must be determined at compile-time:

- If the actual parameter is a string variable or a string constant (or a substring thereof), the target type is the same as the variable's or constant's type, respectively.
- If the actual parameter is a declared string function, the target type is the same as the function's type.
- If the actual parameter is a predefined function returning a string, the target type can always be determined from the target types of its actual parameters (cf. B-10.2.).
- If the actual parameter is a concatenation of several string operands, the target type is a string of length s , where s is the sum of the operands' maximum lengths.

The type adaptation rules for string types (cf. E-2.5.1.) define that strings are truncated to the right if they are longer than the target type. Hence, if, during evaluation of a string expression, the string becomes longer than the expression's target type length, the "overflowing" right part of the string can be ignored. Nevertheless, the string expression must be evaluated to its very end in order to fulfil the rule that expressions be always evaluated completely, even if the evaluation of some parts of it might be redundant (cf. B-4.); the result string of the expression is just not expanded beyond the target type's (i.e. the string temporary's) length.

As a consequence, the PISA compiler can determine and allocate the space needed for string temporaries at compile-time. If the string expression contains function designators with string expressions as actual parameters, additional temporaries are allocated at compile-time. Evidently, the same space can be used for holding string temporaries for nonnested string expressions within a procedure or function as long as the string expressions are not part of the same actual parameter list. The total length of this space is equal to the maximum of the sum of the simultaneously used string temporaries' lengths and is known at compile-time.

E-2.5. Coroutine run-time management

The run-time management of simple procedures is well known and has been described in several publications (see e.g. [Aho/Ullman,77]). The introduction of coroutines in PISA, however, requires a slightly more complex mechanism of handling routine activations. This chapter is intended to give a PISA implementor some help in designing a coroutine implementation suitable for PISA. The programming language used to describe the implementation is an informal dialect of PISA:

- Label variables denote run-time addresses of labels. Goto statements can denote label variables, in which case control is transferred to the address held in the label variable.
- Type declarations may contain a formal parameter list. The actual parameters are substituted for the formal ones when the type is denoted in a declaration.
- The type "area" denotes a data space of undefined structure with a given length.
- Whenever an identifier is enclosed in @ symbols (e.g. @nest@), this construct has to be replaced by a constant of type integer at compile-time. The text explains the correct replacement for this "macro" construct.
- Pointers can also refer to declared (static) variables. The function "addr" returns the address of the declared variable given as actual parameter.

Since at the level of implementation, the run-time management of procedures and functions is the same, we only use the term "procedure" in the sequel. It refers to PISA procedures and PISA functions collectively.

Coroutine descriptors

For every coroutine in a PISA program, a coroutine descriptor is allocated during the entire lifetime of the program. Additionally, a coroutine descriptor is allocated for the main procedure (B-9.1.) and for each external routine (B-9.2.).


```

type cor_descr (maxnest: 1..100) = record
  down, right:    ->cor_descr
  oldest, youngest: ->act_rec
  invoker:        ->cor_descr
  resume:         label
  display:        array(0..maxnest) of ->act_rec
end record

```

The static (textual) coroutine structure of a PISA program or external routine is encoded by a multiway tree containing coroutine descriptors as its nodes. The fields "down" and "right" link the tree nodes as follows: if the coroutine C represented by the node under consideration is declared at the textual coroutine nesting level n, then "down" points to the textually first coroutine Cs such that Cs is enclosed in C and is declared at level n+1. If no such Cs exists, "down" contains the value "nil". If Cf is the coroutine enclosing C at level n-1, "right" points to the coroutine Cb textually immediately succeeding C such that Cb is declared at level n and both C and Cb have the same Cf. If no such Cb exists, "right" points to Cf. If no Cf exists, "right" contains the value "nil".

The field "oldest" points to the oldest activation record (see below) of the coroutine, or it contains "nil" if the coroutine has no activation records allocated. The oldest activation record of a coroutine, if present, is always the activation record associated with the procedure defined by the coroutine itself. The field "youngest" points to the youngest activation record allocated for the coroutine. If there are less than two activation records for the coroutine, "youngest" contains the same value as "oldest".

The field "invoker" points to the coroutine descriptor of the invoking coroutine. "resume" holds the address of the label where execution has to resume after invocation of a suspended coroutine. The components of the array "display" point to the activation records currently accessible according to the scope rules for PISA. The i-th component points to the activation record currently associated with the textual procedure nesting level i, where the main procedure or external routine is at level 1. Level zero may be used for addressing statically allocated objects. The array "display" represents what is called "static chain" in some implementations of block structured languages (e.g. [Ammann,76]). The number of components in the display is given by the deepest textual procedure nesting level in the coroutine's proper text and is known at compile-time. The coroutine's proper text is that part of its text that is not enclosed in nested coroutines.

Activation records

For every procedure invocation, a so called activation record is dynamically allocated.

```

type act_rec (length: 0..10000000) = record
    retaddr:      label
    pred:         ->act_rec
    dispsave:     ->act_rec
    dataarea:     area (length)
end record

```

The label field "retaddr" holds the address of the label where execution is to continue after termination of the procedure. The field "pred" (predecessor) points to the activation record of the invoking procedure within the same coroutine. The field "pred" of the coroutine's oldest activation record contains the value "nil". The field "dispsave" (display entry save) is used to save the actual coroutine's display component overwritten by this procedure. The field "dataarea" denotes the dynamic data space used by the procedure. The length of this data space is known at compile-time.

Statically allocated control variables

For every coroutine, main procedure, and external routine in a PISA program, a coroutine descriptor is statically allocated at a known address:

```

var cd_1: cor_descr (@nest_1@)
var cd_2: cor_descr (@nest_2@)
:
:
var cd_n: cor_descr (@nest_n@)

```

The actual parameters of the type "cor_descr" (@nest_i@) specify the maximum textual procedure nesting level in the coroutine's proper text (see above). Before execution of the PISA program starts, the fields "down", "right", "oldest", and "youngest" are set according to their description given above under "coroutine descriptors". The field "display(0)" of the main procedure's coroutine descriptor and of all the external routines' coroutine descriptors is set to point to their static data area.

The variable "c" points to the active coroutine descriptor at any given time:

```

var c:      ->cor_descr

```

Before a procedure or coroutine is invoked, the calling sequence places the return address into the variable "ret":

```
var ret: label
```

Coroutine prologue

```

1  begin cor_prologue
2      var p:                ->cor_descr
3      var d:                0..100
4      var continue_at:     label

5      p := addr(cd_@i@)
6      p->.invoker := c

7      if p->.oldest = nil then
8          p->.oldest := allocate(act_rec(@length@))
9          p->.youngest := p->.oldest
10         loop copy_display for d := 0 to @nest@-1
11             p->.display(d) := c->.display(d)
12         end loop copy_display
13         p->.display(@nest@) := p->.oldest
14         p->.resume := end_prologue
15         p->.oldest->.pred := nil
16     end if

17     c := p
18     c->.oldest->.retaddr := ret
19     continue_at := c->.resume
20     c->.resume := recursion_trap
21     goto continue_at

22 end begin cor_prologue
23 *> end_prologue:

```

This prologue code is executed whenever a coroutine is invoked. Line 5 sets the address of the coroutine's descriptor in the variable p. If the coroutine has no activation records, lines 8-15 are executed. The "allocate" function designated in line 8 allocates the coroutine's activation record and returns its address. The construct "@length@" has to be replaced by the length of the dynamic data space needed. Lines 10-12 serve to copy the relevant part of the invoker's display. "@nest@" has to be replaced by the textual nesting level of the actual coroutine. Lines 10-12 must be omitted for all coroutines at textual level 1 (main procedure and external routines). Line 14 sets the resume address immediately following the end of the "cor_prologue" compound statement. Coroutines may not be invoked recursively.

Therefore, line 20 sets the resume address to the error routine "recursion_trap". If the coroutine is invoked again before it is suspended or terminated, the error routine gains control.

Coroutine_suspension

```

1  begin suspension
2      var p:      ->cor_descr
3      p := c
4      c := p->.invoker
5      p->.resume := resume_addr
6      goto p->.oldest->.retaddr
7  end begin suspension
8  *> resume_addr:

```

This code is generated by a suspend statement

Coroutine_epilogue

```

1  begin cor_epilogue
2      var continue_at:  label
3      continue_at := c->.oldest->.retaddr
4      deallocate(c)
5      c := c->.invoker
6      goto continue_at
7  end begin cor_epilogue

```

This epilogue code is executed when control reaches the end of the coroutine block's statement part. The procedure "deallocate" deallocates the activation record of the actual coroutine and the ones of all its enclosed coroutines that still contain activation records:

```

1  procedure deallocate (p: ->cor_descr)
2      var p1:          ->cor_descr
3      var p2,p3:       ->act_rec
4      p1 := p
5      loop traverse while p1<>p->.right
6          if p1->.oldest<>nil then
7              p2 := p1->.youngest
8              loop dealloc while p2<>nil
9                  p3 := p2->.pred
10                 free(p2)
11                 p2 := p3
12             end loop dealloc
13             p1->.oldest := nil
14             p1->.youngest := nil
15             if p1->.down<>nil then p1 := p1->.down
16             else p1 := p1->.right end if
17         else
18             p1 := p1->.right
19         end if
20     end loop traverse
21 end procedure deallocate

```

The procedure "free" on line 10 frees the activation record referenced by the actual parameter.

Prologue of simple procedure

```

1  begin simple_prologue
2      var p:          ->act_rec
3      p := allocate(act_rec(@length@))
4      p->.retaddr := ret
5      p->.pred := c->.youngest
6      p->.dispsave := c->.display(@level@)
7      c->.youngest := p
8      c->.display(@level@) := p
9  end begin simple_prologue

```

This prologue code is executed whenever a simple procedure is invoked. The "allocate" function designated on line 3 allocates an activation record and returns its address. The construct "@length@" has to be replaced by the length of the dynamic data space needed. The construct "@level@" must be replaced by the textual procedure nesting level of the actual procedure.

Epilogue_of_simple_procedure

```

1  begin simple_epilogue
2      var p:         ->act_rec
3      var continue_at:  label
4      p := c->.youngest
5      continue_at := p->.retaddr
6      c->.youngest := p->.pred
7      c->.display(@level@) := p->.dispsave
8      free(p)
9      goto continue_at
10 end begin simple_epilogue

```

The procedure "free" on line 8 frees the activation record referenced by the actual parameter. The construct "@level@" has to be replaced by the textual procedure nesting level of the actual procedure.

Procedure_invocation

```

1  begin call
2      ret := return_to
3      goto @procedure@
4  end begin call
5  *> return_to:

```

The construct "@procedure@" has to be replaced with the label (i.e. address) of the procedure invoked. The above program segment models a "branch and link" instruction. Notice that the invocation code is the same for simple procedures and coroutines.

Addressing_of_data

During compilation, all data addresses are compiled to a tuple (t,o), where "t" is the textual procedure nesting level of the object and "o" the offset within the procedure's activation record. Since the addresses of all currently accessible activation records are held on the active coroutine's display, the address of the data object denoted by (t,o) is c->.display(t)+o. If the address of the actual coroutine descriptor (variable c) is held in a register, the address calculation results in loading the address from the display into a base register and using it together with the offset in the instruction that accesses the object. As an optimization, the base addresses of the most frequently accessed scopes may be held in dedicated registers.

E-2.6. Relation tests on real operands

Relation tests on real operands have been introduced in B-4.2.5. This chapter showed that such relation tests are performed after the operands have been rounded to the precision given by their types. If one of the operands is a constant and if this constant is not preciser than the other operand, the PISA compiler may replace the relation test with another (and usually much faster one) using the "rep" option:

```

x < c      =>      x rep< r
x > c      =>      x rep> s
x <= c     =>      x rep<= s
x >= c     =>      x rep>= r
x = c      =>      x rep>=r and x rep<=s
x <> c     =>      x rep< r or x rep> s

```

x is an operand of type "fix" or "float", c is a real constant or an integer constant. r and s define the range of real values that become equal to c when rounded to the precision given by the type of x so that

```

      r < c < s
and  r = lowest  v for which rnd(v,x)=c
and  s = greatest v for which rnd(v,x)=c

```

where "rnd(v,x)" denotes rounding of the value v to the precision given by the type of x. Both r and s can be computed at compile-time as follows:

- If x is of type "fix" with f fractional digits, then:

```

r := c - 10**(-f) / 2
s := c + 10**(-f) / 2
if r<0 then incr(r) end if
if s>0 then decr(s) end if

```

- If x is of type "float" with m mantissa digits and if the constant c is greater than zero, then:

```

s := 10**(trunc(log10(c))-m+1) / 2
r := c - 10**(trunc(log10(c-s))-m+1) / 2
s := c + s
decr(s)

```

- If x is of type "float" with m mantissa digits and a smallest nonzero value of z and if the constant c is equal to zero, then:

```

r := 10**(trunc(log10(z))-m+1) / 2
s := z - 10**(trunc(log10(z-r))-m+1) / 2
r := -s
incr(r)
decr(s)

```

- If x is of type "float" with m mantissa digits and if the constant c is less than zero, then:

```

r := 10**(trunc(log10(-c))-m+1) / 2
s := c + 10**(trunc(log10(-c-s))-m+1) / 2
r := c - r
incr(r)

```

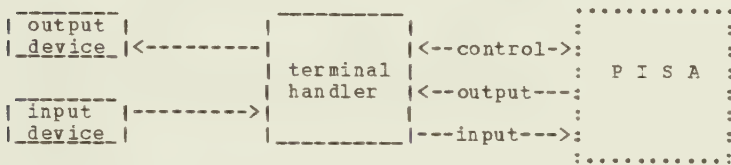
r and s have the same type as x and, hence, the same internal representation as well. The procedure "incr" adds 1 to the least significant digit of its actual parameter's internal representation, "decr" accordingly subtracts one from it. Informally, these procedures shift the internally represented value by one step to the right (incr) or to the left (decr) on the digitized number scale given by their parameter's internal representation.

E-3. THE USER INTERFACE

=====

The operational characteristics of the user interface have been defined in B-11.1. and in C-1.2. This chapter sketches an implementation of the user interface that meets the requirements implied by the operational characteristics.

The user interface consists of three components: the input device, the output device, and the terminal handler. The latter controls the input and output devices and connects them to PISA.



E-3.1. The input device

If the user interface connects a real terminal to PISA, the input device is the terminal's keyboard. If the terminal connected is a pseudo terminal, the input device is a software monitor in Hoare's sense [Hoare,74] which controls a buffer. The buffer receives data produced by the controlling program via the "enter data on pseudo terminal" SSR (see Appendix B-II); the terminal handler is the buffer's consumer.

As a minimal requirement, the input device must be capable of producing the following characters:

1. Alphabetic: The capital letters A to Z
2. Numeric: The digits 0 to 9

3. Special:

!	'	-	=
"	(	.	>
#	)	/	?
\$	*	:	@
%	+	;	
&	,	<	space

4. Control:	<u>function</u>	<u>recommended character</u>
	backspace character	BS (ASCII 8)
	delete line	NAK (ASCII 21)
	horizontal tab	HT (ASCII 9)
	carriage return	CR (ASCII 13)
	line feed	LF (ASCII 10)
	escape	ESC (ASCII 27)
	interrupt	ETX (ASCII 3)
	stall output	DC3 (ASCII 19)
	resume output	DC4 (ASCII 20)

The characters associated with the different control functions are not defined since they depend on the type of terminal available and, sometimes, on the operating system. The characters recommended above are the ones used in DEC's RSTS-E [DEC]. They can be generated on most ASCII terminals and should be used wherever feasible.

The keyboard constituting the input device of a real terminal may provide additional keys, particularly the set of small letters.

E-3.2. The output device

If the user interface connects a real terminal to PISA, the output device consists of a mechanical or electronic unit capable of producing legible sequences of characters. All alphabetic, numeric, and special characters generated by the input device must have a unique representation on the output device. The characters displayed or printed are arranged in output lines. Each output line must be able to hold 80 characters or more. At least the 24 previous output lines should be visible on the output device.

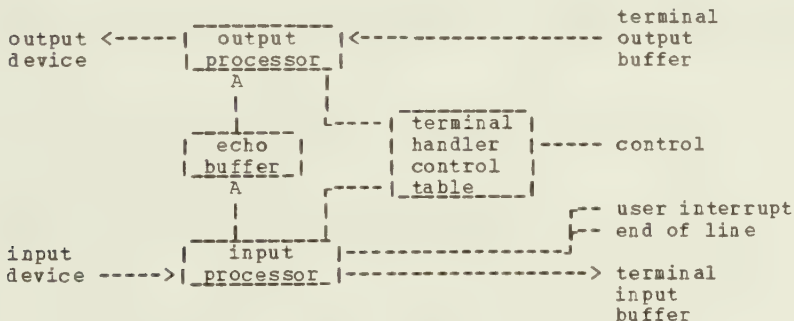
Since almost every terminal is able to display the non-control characters provided on its keyboard, and because display tubes of 24x80 characters or serial printers with a line width of 80 or more characters are used by most of the terminal manufacturers, the above definition of a real terminal's output device is consistent with a "de facto" industrial standard.

If the user interface connects a pseudo terminal to PISA, the output device is a software monitor in Hoare's sense [Hoare,74] which controls a buffer. The buffer receives data produced by the terminal handler; the controlling program is its consumer by accessing it via the "read data from pseudo terminal" SSR (see Appendix E-II).

E-3.3. The terminal handler

The terminal handler is the system component that processes the data transfer between PISA and the input and output devices. It can be implemented by the terminal's hard- or software, by a line adapter, by a controller, by the operating system, by another suitable piece of hard- or software, or by any combination thereof. This chapter merely defines the operation of the terminal handler. In fact, it might turn out that certain of its functions as described in the sequel do not have to be considered for a PISA implementation because they are already present in one of the system's components or are redundant.

The terminal handler may be structured into the input processor, the output processor, the terminal handler control table, and the echo buffer:



TERMINAL

TERMINAL HANDLER

PISA

E-3.3.1. The input processor

The input processor accepts the characters transmitted by the input device and processes them. Its basic operation, as required for proper execution of a PISA implementation, is outlined below and starts with step (1).

- (1) Accept the next input character from the input device.
- (2) Translate the input character from step (1) using the input translation table held in the terminal handler control table. The character received from the translation is called translated input character. If the translated input character is a "null character" (e.g. ASCII 0 or EBCDIC 0), go to step (1).
- (3) If the translated input character is not a control character (i.e., if it is an alphabetic, numeric, or special character), transfer it to the terminal input buffer as well as to the echo buffer and go to step (1).
- (4) If the translated input character is one of the control characters listed below, perform the action indicated and then go to step (1), otherwise go to step (5). The abbreviations used for the control characters are those given in E-3.1. under "4. Control".

BS: If no character precedes BS on the actual line, go to (1). Else set the buffer pointer of the terminal input buffer one character back and transfer the BS character to the echo buffer.

NAK: If the actual line is not empty, delete it from the terminal input buffer. Transfer the character sequence !/</CR/LF to the echo buffer (the slashes visually separate the characters, they are not transmitted).

CR,LF,ESC: Transfer the control character to the terminal input buffer, transfer the character sequence CR/LF to the echo buffer, and signal the end of the actual line to PISA.

ETX: If the "interrupt_enable" flag in the terminal handler control table is false, ignore the ETX, otherwise perform the following steps:

- a) Clear the terminal input buffer and the terminal output buffer.

- b) Set the "echo" flag in the terminal handler control table to true.
- c) Set the "stall_output" flag in the terminal handler control table to false.
- d) Signal a user interrupt to PISA.

DC3: Set the "stall output" flag in the terminal handler control table to true.

DC4: Set the "stall output" flag in the terminal handler control table to false.

- (5) Transfer the character to the terminal input buffer and to the echo buffer, then go to (1).

The input processor, as described above, operates on two output buffers (the echo buffer and the terminal input buffer) and an input device, possibly buffered as well. It, of course, has to synchronize its operation with the other system components (hard- or software) accessing these buffers. If either of the two output buffers is full, the input processor should not accept any more input from the input device. Since most terminals make no provision for locking their keyboard to inhibit further input, a commonly applied procedure is to simply ignore the keyboard input until the output buffers are ready to accept more data.

The input processor transfers characters to the echo buffer only as long as the "echo" flag in the terminal control table is true and the mode field specifies "full_duplex". If any of these two conditions does not hold, all transfer commands to the echo buffer are ignored.

E-3.3.2. The output processor

The output processor reads characters from the terminal output buffer and from the echo buffer, processes them, and passes the proper output code to the output device. The operation of the output processor is shown by the following algorithm which starts with step (1).

- (1) If the echo buffer is not empty, extract the next character from it, make it the output character, and go to step (4).
- (2) If the "stall output" flag in the terminal handler control table is true, then go to (1).

- (3) If the terminal output buffer of PISA is not empty, extract the next character from it and make it the output character, else go to (1).
- (4) If the output character is a HT (horizontal tab) and the "hardware tab" flag in the terminal handler control table is false, replace the HT by as many spaces as are necessary to move the printing head or cursor to the next tab stop (see E-3.3.3.). If the field "tab_step" of the terminal handler control table contains zero, ignore the HT and go to step (1).
- (5) Translate the output character using the output translation table and transfer the translated output character to the output device. The output translation table is held in the terminal handler control table.
- (6) If the output character was not a control character go to step (1), else transfer as many "null characters" (e.g. ASCII 0 or EBCDIC 0) to the output device as are necessary to idle the delay time specified for the control character, then go to step (1). The delay time and the transfer rate, together determining the required number of "null characters", are both held in the terminal handler control table (see E-3.3.3.).

The output processor, as described above, operates on two input buffers (the echo buffer and the terminal output buffer) and an output device, possibly buffered as well. Therefore, it has to synchronize its operation with the other system components (hardware or software) accessing these buffers. For the sake of simplicity, the operations of steps (1), (2), and (3) have been outlined to constitute a "busy wait" looking for nonempty buffers. In practice, however, these steps could be driven by some signalling mechanism to overcome the disadvantages of busy waits.

Step (6) of the output processor introduces a delay time generator. Delay times are required to wait for the completion of mechanical or electronic operations triggered by certain control characters. Older printing terminals that have no internal buffer incorporated into their output device need delay times for carriage returns, line feeds, tabs, etc. Some terminals of recent origin providing editing features like protected fields, video control, cursor addressing, etc., are controlled by a built-in microprocessor. They need a certain time, in the magnitude of several milliseconds, to complete specific operations as switching from formatted to unformatted mode, clearing nonprotected fields, etc. During this time, all

or part of the characters received are ignored. The delay time generator makes sure that the lost characters are just "null characters". If the terminal provides full internal buffering to hold all characters received during the longest delay and even at the highest transfer rate, the delay time generator is obsolete.

E-3.3.3. The terminal handler control table

The terminal handler control table serves to control certain operations of the terminal handler's input and output processors. Each active user interface in a PISA system (i.e. each active session) has its own terminal handler control table. It is a record variable of the following type:

```
var thct = record
    input_xlate_tab:    string(256)
    output_xlate_tab:   string(256)
    delay_tab:          array(cc) of 0..1000
    term_type:          (pseudo,crt,
                        storage_tube,printing)
    line_width:         1..255
    page_size:          0..255
    transfer_rate:      0..10000
    hardware_tab:       boolean
    tab_step:           0..255
    mode:               (half_dup,full_dup)
    echo:               boolean
    stall_output:       boolean
    interrupt_enable:   boolean
end record
```

The fields and their functions are:

The input translation table is used by the input processor to translate the characters transmitted by the input device. Translation is defined by

$t := \text{xlate}(i, \text{thct.input_xlate_tab})$ (cf. E-10.2.)

where i is the character received from the input device and t the translated input character.

The output translation table is used by the output processor to translate the characters before they are transmitted to the output device. Translation is defined by

`o := xlate(c,thtc.output_xlate_tab)` (cf. B-10.2.)

where `c` is the output character and `o` the character transferred to the output device.

The delay table specifies for each control character the idle time (in milliseconds) required after it has been transmitted to the terminal.

The terminal type indicates whether the terminal connected to the user interface is a pseudo terminal, a cathode ray tube terminal, a storage tube terminal, or a printing terminal.

The line width contains the physical line width of the output device.

The page size specifies the number of lines the output device can hold on one page. A page size of zero indicates an unlimited page size (e.g. for printing terminals).

The transfer rate indicates the number of characters that are transferred to the terminal in one second (not bauds!).

If the hardware tab flag is true, the output device is assumed to have a built-in tabulating mechanism. If the flag is false, the terminal does not provide such a feature.

The tab step defines the step-width of tabulations so that the n -th tab stop is assumed at column $n \cdot \text{tab_step}$, where the leftmost column of an output line is column 1.

The mode field indicates whether the terminal connected operates in full-duplex (remote echo) or half-duplex (local echo) mode.

The echo flag specifies whether or not the characters received from the input device of the terminal should be echoed back to its output device (echo = true means produce echo).

As long as the stall output flag is true, the output processor does not transfer any characters from PISA's terminal output buffer to the output device.

If the interrupt enable flag is true, the input processor may signal a "user interrupt" to PISA, else it has to ignore all user interrupts.

The terminal handler control table has to be properly initialized for every terminal connected to PISA. The implementation handbook must specify the default characteristics assumed for real and pseudo terminals.

The fields "input_xlate_tab" to "mode" in the terminal handler control table can be set by any PISA program with the SSR 1006 (see Appendix B-II). The fields "echo" and "interrupt_enable" are controllable by predefined procedures introduced in B-11.1.

E-4. THE SYSTEM INTERFACE

=====

The system interface of a PISA program has been informally explained in B-11.2. as a message handler processing the data transfer between the PISA program and the run-time system. The system service requests (SSRs) used to communicate with this message handler have been introduced in B-11.2. and described in detail in Appendix B-II.

The implementation of the system interface is presumably that part of the entire PISA system revealing the most remarkable conceptual differences from implementation to implementation. Its internal structure is almost completely based on the supporting operating system. Therefore, hardly any general guidelines for the system interface's implementation can be given. On some systems, the system service requests may be compiled straightforward into calls to operating system service routines, while on others rather complex intermediate message handlers are required.

When new system service requests are introduced, their design should follow the one of the requests defined in Appendix B-II. This particularly implies that all recoverable errors during execution of the system service request are to be reported to the PISA program in a similar way as, e.g., in the "open standard sequential file" SSR (SSR 4010 in Appendix B-II).

Appendix E-I: Keyword Index

=====

activation records	F-2.5.
addressing of data	F-2.5.
C-code	E-2.1.
compiler	E-1.
coroutine descriptors	E-2.5.
coroutine management	E-2.5.
data types	E-2.3.
grammar	E-1.1.
input device	F-3.1.
interpreter	E-2.1./E-2.2.
output device	E-3.2.
partial recompilation	F-1.2.
procedure management	E-2.5.
real relation tests	F-2.6.
recompilation	E-1.2.
relation tests	F-2.6.
SLR(1) grammar	F-1.1.
string expressions	E-2.4.
suspension	E-2.5.
system interface	E-4.
T-code	E-2.2.
target type	E-2.4.
terminal handler	F-3.3.
types	E-2.3.
user interface	E-3.

Appendix E-II: References

=====

- [Aho/Ullman,77] Aho V.A. and Ullman J.D., "Principles of Compiler Design", Addison-Wesley, Reading, 1977, pp.350-381.
- [Ammann,76] Ammann U., "On Code Generation in a PASCAL Compiler", Institut fuer Informatik, ETH Zuerich, 1976.
- [DEC] Digital Equipment Corporation, "RSTS-E System User's Guide", Maynard, Massachusetts.
- [DeRemer,71] DeRemer F.L., "Simple LR(k) Grammars", CACM, Vol.14, No.7, 1971, pp.453-460.
- [Hoare,74] Hoare C.A.R., "Monitors: An Operating System Structuring Concept", CACM, Vol.17, No. 10, 1974, pp.549-557.
- [LaLonde et al.,71] LaLonde W.R., Lee E.S., and Horning J.J., "An LALR(k) Parser Generator", Proc. IFIP Congress 71, North-Holland, Amsterdam, 1971, pp.153-157.

SECTION F

Portability

According to Poole and Waite [Poole/Waite,73], "Portability is a measure of the ease with which a program can be transferred from one environment to another: If the effort required to move the program is much less than that required to implement it initially, then we say that it is highly portable." Remarkably, this definition does not introduce an absolute measure of portability but a relative one: The ratio between the work involved in porting an existing program and the work needed to write it from scratch. By no means should the term "portable program" be interpreted as a program that can be moved to another environment with a null effort. What we are interested in is to minimize the costs of transferring programs.

To complete the description of the programming system PISA, this section discusses the portability of PISA programs between different PISA implementations. Although there are always applications where portability of programs is of no concern or only of minor importance, a programming language designed to yield highly portable programs has several benefits in medium to long term commercial reasoning:

- Even when software is not intended to be distributed to other implementations, there might come the day when the old computer system has to be replaced. To find the optimal replacement for the old system, a cost-benefit analysis of the different alternatives is usually performed. The cost to port the software from the old computer to the new one enters the cost-benefit analysis as initial investment in the new system, an investment that has to be depreciated during a given period. If the portability cost is low for all systems in the evaluation process, the other evaluation criteria get a greater relative weight. Today we quite often face the situation that a change to another computer series is desirable from all viewpoints, yet the change is not affordable because it is too costly to adapt or rewrite the existing programs. A first consequence of a portable language is therefore an augmented flexibility for computer evaluation.

- Standard software packages that are available and useful for a certain application can often not be used because they are not portable to the desired computer system. This holds both for licenced software and for programs available free of charge from several institutions, especially in the academic environment. Hence, a second benefit of a portable language is the facilitated trade in software.
- A programmer experienced in a highly portable language does not need a specific training to get used to a language implementation's peculiarities when he starts working with this implementation. Using a portable language results in less educational expenses when new programmers who know the language are employed or when a new language implementation (i.e., a new computer system) replaces the old one.

These reasons seem enough justification for designing a portable language and for providing utilities enhancing the program's portability. The following three chapters describe the guiding principles that make PISA programs highly portable according to Poole's and Waite's definition.

Table of contents

DESIGN OF A PORTABLE LANGUAGE	F-1.
PISA as a highly portable language	F-1.1.
Basic vocabulary	F-1.1.1.
Data types	F-1.1.2.
Relation tests on real operands	F-1.1.3.
Interfaces of a PISA program	F-1.1.4.
Implementation parameters	F-1.2.
THE PORTABILITY CHECKER	F-2.
Lexical structure	F-2.1.
Data types	F-2.2.
Constants	F-2.3.
Expressions	F-2.4.
Statements	F-2.5.
Declared procedures and functions	F-2.6.
Predefined procedures and functions	F-2.7.
System service requests	F-2.8.
VALIDATION OF PISA IMPLEMENTATIONS	F-3.
Appendix F-I: Keyword index	
Appendix F-II: References	

F-1. DESIGN OF A PORTABLE LANGUAGE

=====

The definition of the programming language PISA introduces an abstract, mathematical model of computation not related to any specific implementation. A syntax for an algorithmic language is given and precise semantics associated with all valid constructs of this language. It is the implementor who "constructs" a real PISA-machine based on the abstract PISA-model. This machine should be the operationally equivalent physical appearance of the model, i.e., any input produces exactly the same output on both the machine and the model. Consequently, all distinct implementations (or machines) have precisely the same operational characteristics and PISA programs are portable without any adaptation effort at all.

As with most (or probably all) high level languages, this goal cannot be reached with PISA. The reasons for the operational discrepancy between the model and its implementation are:

1. The model does not consider all implications of the finite resources given by an implementation. The implementation always imposes several restrictions on the model's definition. More common examples for this are deepest textual nesting level, maximum number of identifiers, largest program size at execution time, etc.
2. The model defines abstract data types and the operations on them, whereas the implementation has to define a suitable digital representation for the types and must perform the operations given by the model on the hardware available. Because the hardware architecture differs considerably between some computer series, the representation of data types and the operations on the types is not unique among all implementations of a language.

It is not so much the first reason making it bothersome and difficult to port a program; its effect is quite clear: either the program can be compiled and executed, or its compilation or execution is terminated at some point. The effects of the second reason above can be much more sneaking: the program might execute and come to a normal end yet produce results different from those obtained on the original implementation. The reasons for this can be manifold: other character set, data type representation that differs from the one anticipated by the

program for performing input/output, incompatible arithmetic (especially on floating point numbers), to name but a few.

The portability of PISA programs is enhanced in three ways:

- The language PISA has been designed as a highly portable programming language. This aspect of portability is discussed in F-1.1.
- Every implementation of PISA must specify a set of parameters defining the few implementation dependent language rules. The implementation parameters will be introduced in F-1.2.
- A special program called "portability checker" accepts the implementation parameters of the "source" and of the "target" implementation together with the PISA program to port from "source" to "target" and produces a set of portability diagnostic messages. The portability checker is described in F-2.

F-1.1. PISA as a highly portable language

As mentioned before, designing a programming language means designing an abstract model of computation without any reference to a specific implementation. This is certainly the only way a modern high level programming language should be defined [Wirth,76]. When the language's abstract concepts are designed, however, extreme care has to be taken to ensure that the abstract concepts be implementable on a digital computer with exactly the syntax and semantics given. If a language is defined without any consideration of the physical properties of the class of machines designated to execute the language, it is always the implementor adding rules to the language, rules that alter the abstract model.

The design of PISA has been guided by the trial to keep the abstract PISA model of computation independent of any physical implementation, yet to build the model well aware of the characteristics of today's digital computers.

F-1.1.1. Basic vocabulary

All PISA symbols are composed of characters that are in both the ASCII and in the EBCDIC character set. The first and more obvious implication of this restriction is that PISA programs ported between ASCII and EBCDIC implementations do not need a transliteration because the basic vocabulary can be implemented as defined by the language. The second consequence of the basic vocabulary's restricted character set is related with the alphabet of natural languages introducing special characters like accents, tildes, modifications of vowels, etc. Practical observation shows that the internal codes used for these nonstandard characters are primarily the ones originally associated with {, }, [, and]. A program of a language making use of these brackets looks extremely ugly and unreadable when written on a printer with these characters replaced.

The drawback of the restricted basic character set used by PISA is the necessity of two kinds of twin-character pairs of brackets: "<:" and ">", "(:" and ":)" (cf. B-3. and B-4.1.). PISA's comment brackets "(*" and "*)" have already been used in other languages, including several Pascal dialects.

F-1.1.2. Data types

The main requirement to portable type declarations is that every type define the values of its type. Consequently, types like Pascal's "integer" and "real" have to be rejected for it is the implementation that defines the type's values (range, precision), not the language. PISA does only allow integer subrange type declarations and requires the specification of the desired precision and range for real types (cf. B-2.2.1.2. and B-2.2.2.). The implementation either associates a suitable internal representation with such a type or rejects it as not representable.

The maximum length of string types is implementation dependent. A minimum length of 255 characters, however, has to be supported by every implementation. 255 has been chosen because this length is sufficient for the bulk of string types in practical applications, and because strings up to a length of 255 characters can be handled efficiently on most computers.

The maximum number of elements in a set type is defined by the implementation. Every implementation, however, has to support

sets with up to 256 elements. These definitions ensure that the supported set size is large enough for common set applications (including writing a PISA compiler), but they still allow an implementation to introduce larger sets.

F-1.1.3. Relation tests on real operands

As defined in B-4.2.5., a relation test on two real values is performed after the value of both operands has been rounded to the precision given by the operands' types. If the internal representation of a real value is taken for the relation test without rounding, the result of the relation test is implementation dependent. If, for example, one implementation represents fix type values in decimal format, a test on equality is true when the two decimally represented values are equal; on another implementation using binary floating-point representation for the fix type, the same test on equality is hardly ever true because of the fuzz (or noise) inherent to binary floating-point arithmetic. Similar problems arise with relation tests on float values since the fuzz of the value is determined by the hardware's floating-point arithmetic and, hence, is implementation dependent.

Although PISA's definition of real types and the operations on them do not provide an absolute guarantee for a fully portable real arithmetic, the portability of PISA's real arithmetic is several orders of magnitude higher than, e.g., the one of a Pascal program. An introduction to the problems involved with real arithmetic is given in [Knuth, 69].

PISA introduces the "rep" option in relation tests on real operands specifying that the relation test should be performed on the operands' values as internally represented (cf. B-4.2.5.). This feature is highly undesirable from the viewpoint of a portable language design. It is a tribute to efficiency requirements where rounding is too costly and not essential for a specific relation test. The result of relation tests with the "rep" option, of course, is implementation dependent.

F-1.1.4. Interfaces of a PISA program

The two interfaces between a PISA program and its outside world are the "user interface" (B-11.1.) and the "system interface" (B-11.2.). The user interface is defined completely implementation independent and, hence, the procedures and functions for handling the user interface are fully portable.

The system interface is accessed by procedure statements invoking the "sys" procedure. The first actual parameter is a constant specifying the system service request code. The classification of these codes (cf. B-11.2.) allows to distinguish between portable, semiportable, and nonportable system service requests.

F-1.2. Implementation parameters

Every PISA implementation specifies a set of parameters that define the implementation dependent language parts. These parameters may be represented by the following variable:

```
var impl: record
  integer_low:      number
  integer_high:     number
  charset:          string(256) vary
  fix_low:          number
  fix_high:         number
  fix_fracdig:      1..1000
  fix_totaldig:     1..1000
  float_low:        number
  float_high:       number
  float_mantdig:     1..1000
  float_smallest:   number
  string_maxlen:    1..10000000
  set_maxsize:      1..10000000
  extid_maxlen:     1..16
  extid_char:       string(256) vary
end record
```

The type "number" is a string (in ASCII character code) holding the textual representation of an integer constant (cf. E-2.2.1.2.) or a real constant (cf. B-2.2.2.):

```
type number = string(50) vary
```

This representation of numbers is completely independent of any hardware arithmetic and ensures full portability of the implementation parameters.

The following implementation dependent PISA language constructs are defined by the fields in the implementation parameter record:

- The lowest and highest integer values are given by the fields "integer_low" and "integer_high". The maximum number of elements in an enumerated type is impl.integer_high+1.
- The implementation's character set is defined by the field "charset" such that

```
len(impl.charset) = ord(char,last(char))+1
```

and

```
impl.charset<:i+1,1:> = ascii(val(char,i))
                        for 0≤i≤ord(char,last(char))
```

if "char" is the predefined type "char" of the implementation. The function "ascii" returns the ASCII character code for the character given as actual parameter or, if the character does not exist in ASCII, a DEL character (ASCII 127). If the implementation's character set is the standard ASCII code, the field "charset" contains the string 'ASCII' (in ASCII code), if its character set is EBCDIC, "charset" contains the string 'EBCDIC' (in ASCII code).

- The fix types are parameterized by the lowest and the highest valid fix values (fields "fix_low" and "fix_high"), by the maximum number of fractional decimal digits ("fix_fracdig"), and by the maximum number of decimal digits in the fractional and integer part together ("fix_totaldig").
- The float types supported by an implementation are characterized by the lowest and the highest valid float values (fields "float_low" and "float_high"), by the maximum number of decimal digits in the mantissa ("float_mantdig"), and by the smallest representable positive nonzero float value ("float_smallest").
- The maximum length of string types is given by the field "string_maxlen".

- The field "set_maxsize" defines the maximum number of elements in a set type.
- The implementation may impose restrictions on main procedure identifiers and external routine identifiers (cf. B-9.). The fields "extid_maxlen" and "extid_char" specify the maximum length of these identifiers and the characters allowed in them. A character is legal if it is contained in the field "extid_char", illegal otherwise. The field "extid_char" contains the valid characters in ASCII code.

The finite nature of the resources used by a PISA implementation is the reason for several additional restrictions to PISA's abstract language definition: there is always a maximum program size, a maximum number of identifiers in a program, a maximum textual nesting depth, etc. Restrictions of this kind have not been included in the implementation parameters because, (1) it is quite unlikely that a PISA program is uncompileable due to a lack of resources, (2) a shortage of execution time resources can hardly be detected without actually executing the program, and (3) a lack of resources never causes a hidden misbehaviour of a PISA program - either it can be compiled and executed or an unambiguous error occurs.

F-2. THE PORTABILITY CHECKER

=====

If a PISA program that runs correctly on an implementation has to be ported to another PISA implementation, it can be checked for portability with the "PISA portability checker". This checker analyses the PISA program based on the implementation parameters of the "source implementation" (the one the program has been used on) and of the "target implementation" (the one the program is to be ported to). The output of the portability checker is a set of so called "portability diagnostic messages" designating constructs of the program not supported on the target implementation and revealing other constructs that may have different semantics on the source implementation and on the target implementation.

The following chapters describe the tests performed by the portability checker by giving examples for the portability diagnostic messages and explaining the related checks. It becomes quite obvious that the portability checker is very similar to the PISA compiler except that it does not produce code but portability messages. A practical implementation of the portability checker, therefore, is presumably obtained from a modified PISA compiler.

The portability checker assumes a correct and complete PISA implementation on both the source and the target system. The program to be ported is considered to be syntactically and semantically correct for the source implementation. The portability checker does not check the program for PISA dialects and language extensions.

F-2.1. Lexical structure

Inconsistent character set for PISA vocabulary

One or more of the characters used in PISA's basic vocabulary (B-1.1.) have a different representation on the source implementation and on the target implementation.

No small letters on target implementation

The character set of the target implementation does only define capital letters while the one of the source implementation defined both small and capital letters.

F-2.2. Data types

Too many values in enumerated type

"ord(t,last(t))" for the enumerated type t is greater than the highest integer supported by the target implementation.

Illegal integer subrange type

The integer subrange type defines values outside the range of integer values supported by the target implementation.

Inconsistent character subrange type

The character subrange type does not include the same characters in the same ordering on the source implementation as on the target implementation.

Illegal range of fix values

Fix values of the declared range cannot be represented on the target implementation.

Too many fractional digits in fix type

The target implementation is not capable of representing fix values with as many decimal fractional digits as declared for the fix type.

Too many digits in fix type

The total number of decimal digits resulting from the fix type's range and fractional digits declaration exceeds the maximum given by the target implementation.

Illegal range of float values

Float values of the declared range cannot be represented on the target implementation.

Too many mantissa digits in float type

The target implementation is not capable of maintaining as many decimal mantissa digits as declared for the float type.

Nonzero float value too small

The float type definition specifies a nonzero value that is smaller than the target implementation is able to represent.

String type too long

The length declared for the string type exceeds the maximum string length given by the target implementation.

Inconsistent tag type for variant part

The type of the tag specified in a record definition is of root type "char" and does not include the same set of characters on the source implementation as on the target implementation.

Inconsistent character range in when list

The when list contains a when item of the form $m..n$, where m and n are constants of root type "char". The set of characters defined by $m..n$ is not the same on the source implementation as on the target implementation.

Too many elements in set

The set type contains more elements than supported by the target implementation.

Character subrange type adaptation may fail

Type adaptation from a "char" subrange type t to a char subrange q is required but type t defines characters not in q .

The portability checker does not test the program for representation dependent usages of data types. A program relies on a specific type representation when files are accessed that have been created by non-PISA programs and (illegally) when it denotes field variables that are not in the actual variant of the record.

F-2.3. Constants

Inconsistent character range in value constructor

The value constructor for a structured constant of a set type contains a constant item (B-3.1.) of the form `m..n`, where `m` and `n` are constants of root type "char". The set of characters defined by `m..n` is not the same on the source implementation as on the target implementation.

Illegal integer constant

The integer constant specifies a value that is outside the integer range supported by the target implementation.

Illegal real constant

The real (fix or float) constant specifies a value not representable on the target implementation. It could have too many digits, lie outside the supported range, or be too close to zero.

String constant contains illegal character(s)

The string constant contains one or more characters that are not in the target implementation's character set.

F-2.4. Expressions

Potential character range inconsistency in value constructor

A value constructor for a set of characters contains a range specification `m..n`, where `m` and `n` are expressions of type character. This message is only produced if the source implementation's character set differs from the one of the target implementation.

Relation test on internal real values

A relation test on two real values has been specified with the "rep" option (B-4.2.5.).

Potential string/char relation test inconsistency

A relation test "<", ">", "<=", or ">=" is specified on string or char types. This message is only produced if the source implementation's character set differs from the one of the target implementation.

The portability checker does not attempt to locate expressions that might cause an overflow of any kind when evaluated on the target implementation.

F-2.5. Statements

Inconsistent type of case expression

The type of the expression in a case statement is of root type "char" and does not include the same set of characters on the source implementation as on the target implementation.

Inconsistent character range in when list

The when list contains a when item of the form `m..n`, where `m` and `n` are constants of type "char". The set of characters defined by `m..n` is not the same on the source implementation as on the target implementation.

Potential loop range inconsistency

The loop variable specified in the "for" construct is of root type "char" and the source implementation's character set is not the same as the one of the target implementation.

F-2.6. Declared procedures and functions

Invalid procedure or function name

The name of the main procedure, of an external procedure, or of an external function is syntactically illegal on the target implementation.

All declarations of external procedures and external functions are recapitulated following the portability diagnostic messages.

F-2.7. Predefined procedures and functions

Potential inconsistency of ordinal function

An ordinal function (B-10.1.) is used on a type of root type "char" and the source implementation's character set differs from the one of the target implementation.

Inconsistent "xlate" function

The predefined function "xlate" is not portable between implementations with unequal character sets.

Function "small" is not supported

The predefined function "small" is not supported on the target implementation because the target implementation's character set does not include the set of small characters.

Non-standard predefined procedure or function

A procedure or function has been designated for which no declaration exists in the actual scope and which is not one of the standard predefined procedures and functions (B-10., B-11.).

F-2.8. System service requests

All system service requests contained in the program processed by the portability checker are recapitulated at the end of the portability diagnostic messages. The printout consists of a cross-reference of system service request codes and line numbers of the corresponding system service requests. This information allows to quickly locate the system service requests that have to be replaced or modified.

F-3. VALIDATION OF PISA IMPLEMENTATIONS

=====

The prerequisite for portable PISA programs are correct and complete PISA implementations. All parts of the language PISA must be implemented exactly as defined by Section B and by the implementation's parameters (F-1.2.). Presumably, a PISA implementor makes all his software subject to extensive tests to verify its correct operation. Such tests, however, have the deficiency that they are usually set up by the implementor himself and, hence, reflect the same subjective interpretation of the PISA definition as the implementation: If the implementation is incomplete, the test does probably not contain the data to reveal this incompleteness; if a language rule has been misunderstood and implemented incorrectly, the chance that it is tested for the incorrect semantics is almost certain. It would be of great benefit to have an objective validation system for PISA implementations that either reports inconsistencies between the implementation and the PISA definition or guarantees a correct and complete PISA implementation with a reasonably high degree of certainty.

Such a validation system was designed, implemented, and used by the U.S. Air Force to check COBOL compilers for consistency with a given standard [Hicks,69]. Wichmann proposes a set of test programs to validate a Pascal implementation [Wichmann,78]. After completion of a first PISA implementation, a validation system for PISA should be produced based on a careful selection of all tests done during the implementation work. This validation system should then become available to everybody using or implementing a PISA system who is willing to maintain a feedback path to the institution in charge of the validation system, a feedback consisting of amendments and corrections to the validation system. The result is expected to be a steadily improving and comprehensive validation system for PISA.

The PISA validation system may be programmed as a sequence of PISA programs testing the implementation via one or more pseudo terminals. All elements of the PISA system and the language PISA are validated by systematically entering input on the pseudo terminal and verifying the output received. If any inconsistencies are detected, they are analysed and reported to the terminal connected to the validator's user interface. The validation of the language PISA is done with respect to the implementation's parameters (F-1.2.).

To keep the PISA validation programs fully portable, they should:

- only contain string types up to a length of 255,
- not have set types with more than 256 elements,
- only define very small integer subrange types, say -10000..+10000,
- not be written with respect to any specific character set,
- not contain any real types, and
- not denote any external routines.

Appendix F-I: Keyword index

=====

basic vocabulary	F-1.1.1./F-2.1.
character set	F-1.2./F-2.
data types	F-1.1.2./F-2.2.
external names	F-1.2./F-2.6.
fix types	F-1.2./F-2.2.
float types	F-1.2./F-2.2.
implementation parameters	F-1.2.
integer	F-1.2./F-2.2.
interface	F-1.1.4.
lexical structure	F-2.1.
portability checker	F-2.
real types	F-1.1.3./F-1.2./F-2.2.
relation test	F-1.1.3./F-2.4.
set types	F-1.2./F-2.2.
string types	F-1.2./F-2.2.
system interface	F-1.1.4./F-2.8.
types	F-1.1.2.
user interface	F-1.1.4.
validation	F-3.
vocabulary	F-1.1.1./F-2.1.

Appendix F-II: References

=====

- [Hicks,69] Hicks H.T., "The Air Force COBOL Validation System", Datamation, Aug.1969, pp.73-81.
- [Knuth,69] Knuth D.E., "The Art of Computer Programming", Vol.2, Addison-Wesley, Reading, 1969, pp.180-229.
- [Foole/Waite,73] Poole P.C. and Waite W.M., "Portability and Adaptability", in: Bauer F.L., "Advanced Course on Software Engineering", Lecture Notes in Economics and Mathematical Systems, No.81, Springer-Verlag, Berlin, 1973, pp.183-277.
- [Wichmann,78] Wichmann B.A., "A Pascal Test Suite", National Physical Laboratory, Teddington, Middlesex, 1978, unpublished.
- [Wirth,76] Wirth N., "Programming Languages: What to Demand and how to Assess them", Institut fuer Informatik, ETH Zuerich, 1976.

Offsetdruck
GmbH & Co. KG.

86 Bamberg

Hohe-Kreuz-Str. 9b

M. SCHADEL

• Tel. 0951/54414 • 58461 •